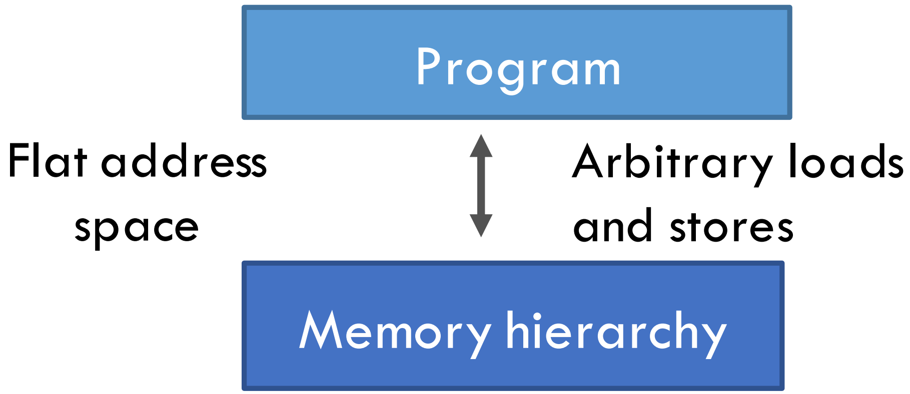
Rethinking the Memory Hierarchy for Modern Languages

Po-An Tsai, Yee Ling Gan, and Daniel Sanchez

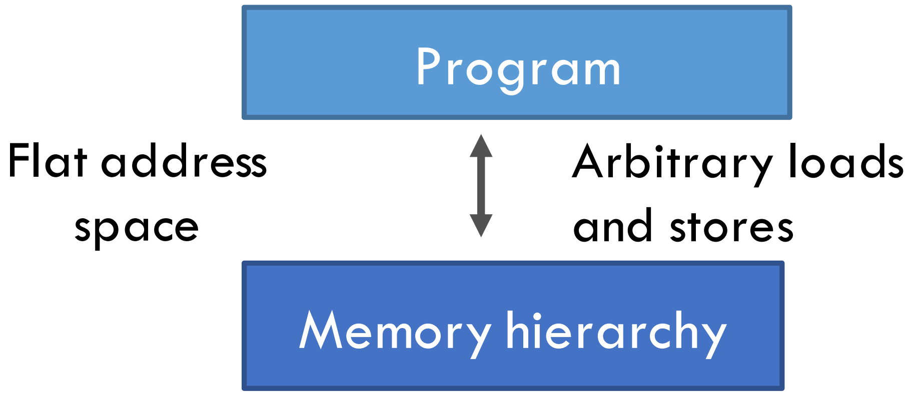
MIT CSAIL

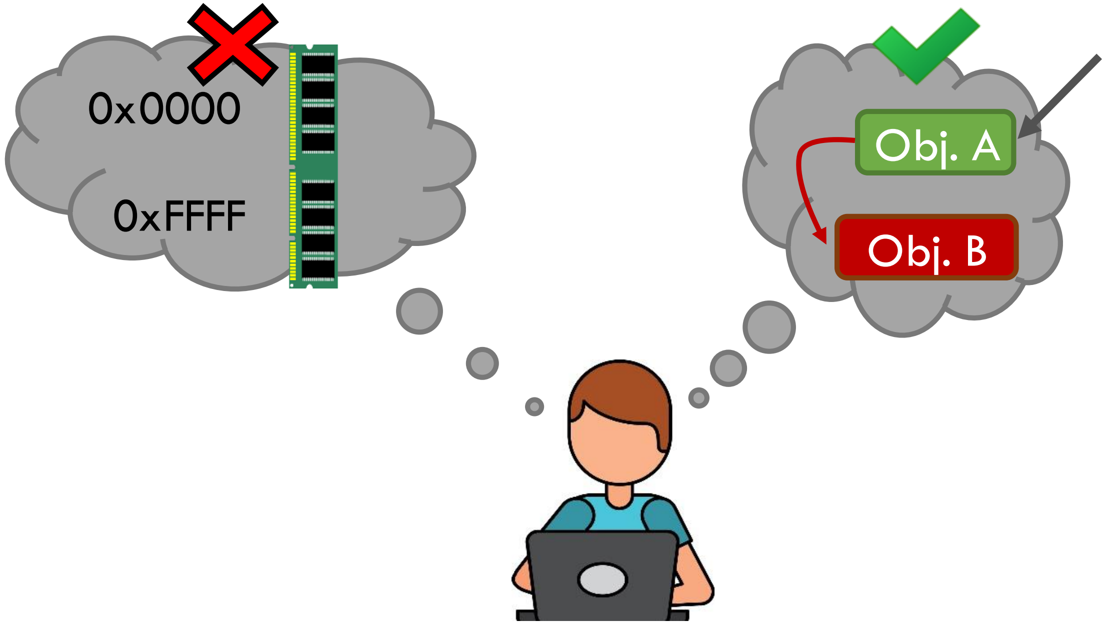
Memory systems expose an inexpressive interface

Memory systems expose an inexpressive interface



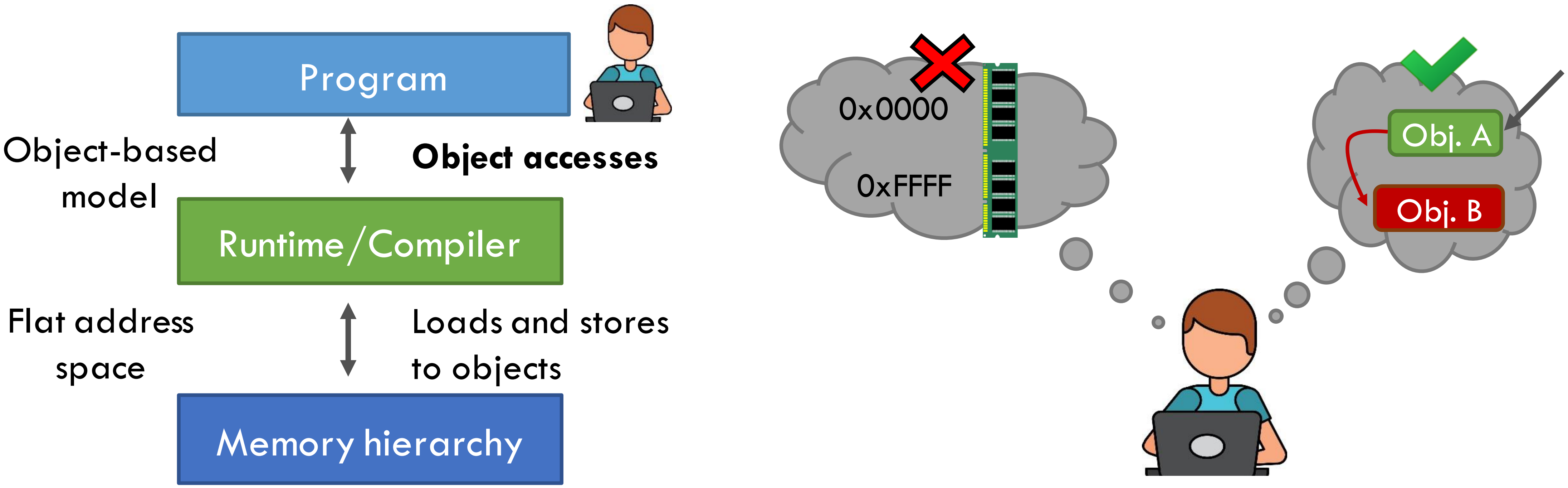
Memory systems expose an inexpressive interface





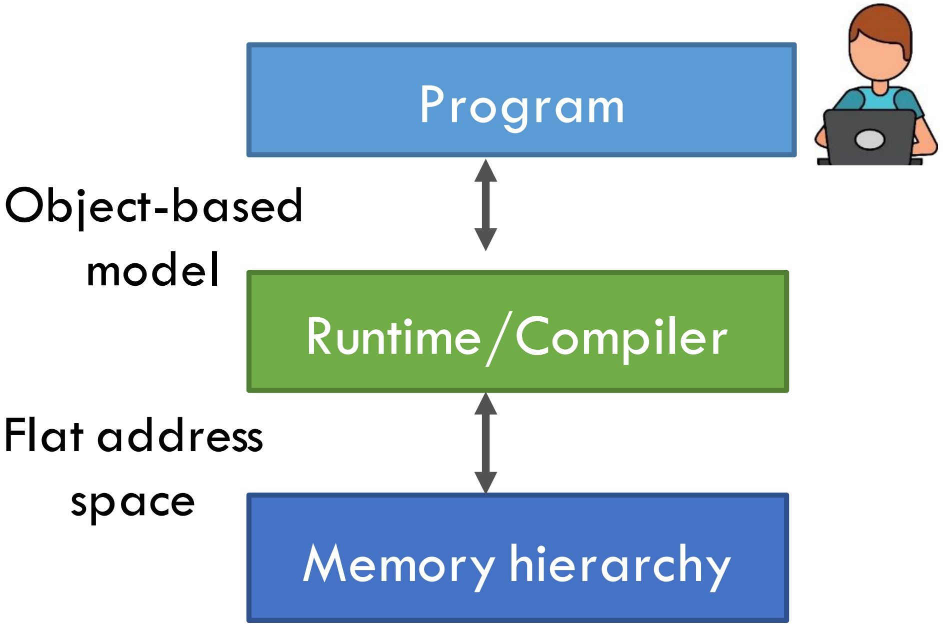
Programmers think of objects and pointers among objects

Modern languages expose an object-based memory model

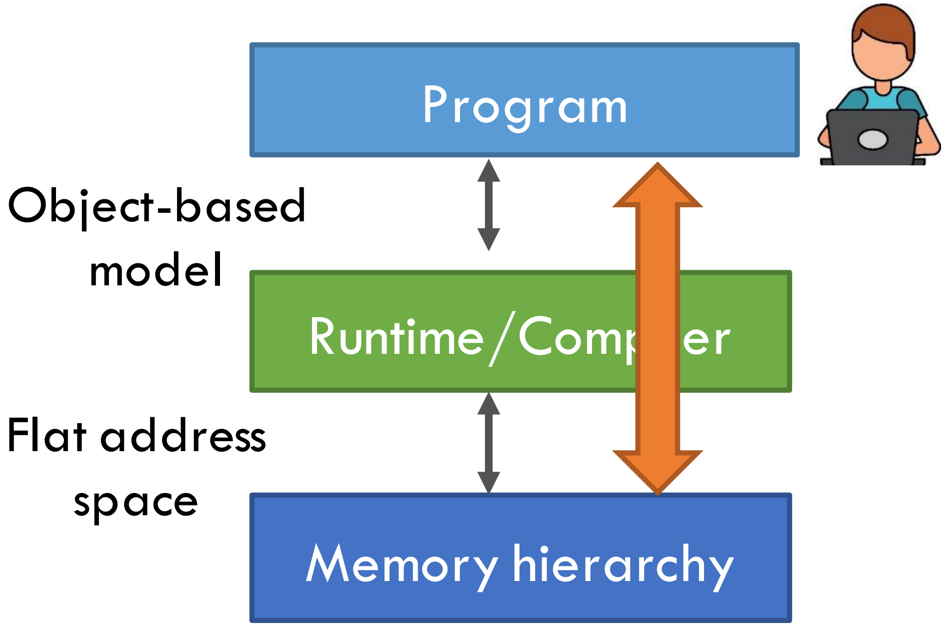


- Strictly hiding the flat address space provides many benefits:
  - Memory safety prevents memory corruption bugs
  - Automatic memory management (garbage collection) simplifies programming

The inexpressive flat address space is inefficient

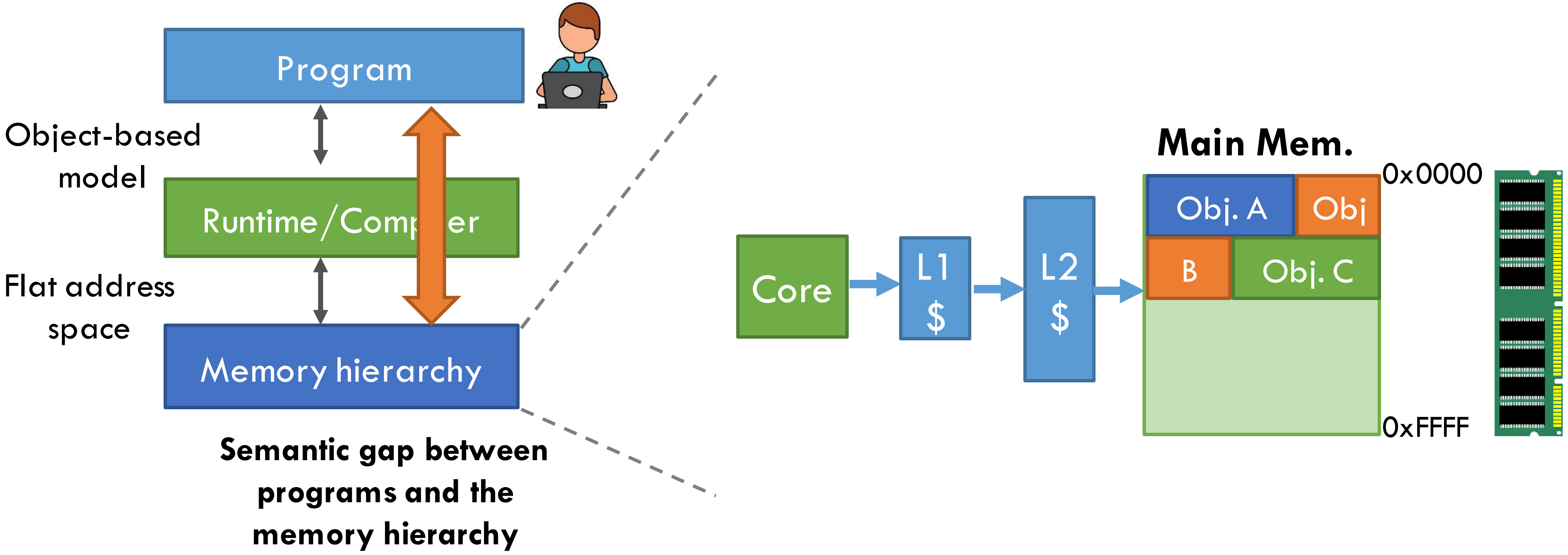


The inexpressive flat address space is inefficient

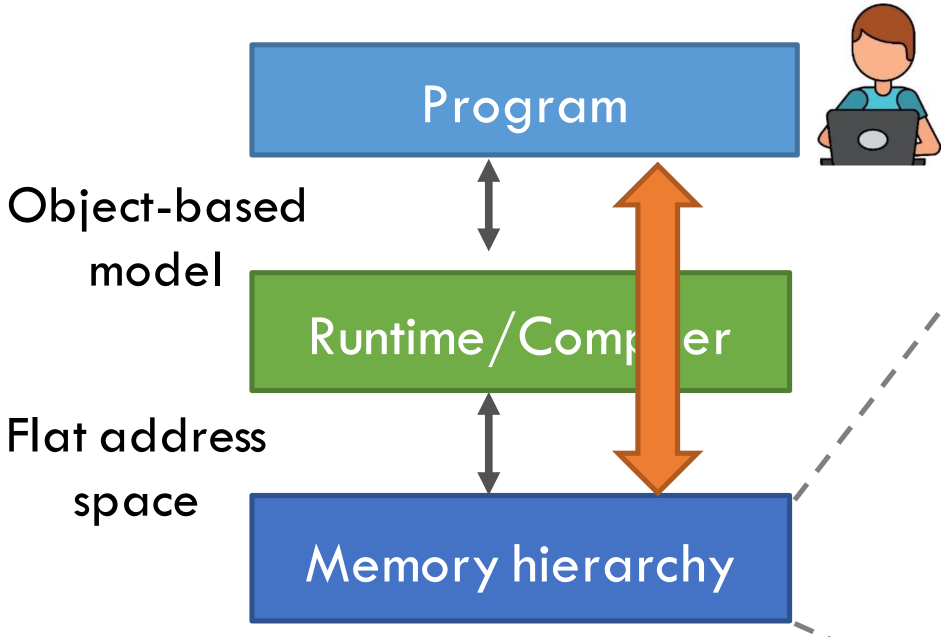


Semantic gap between programs and the memory hierarchy

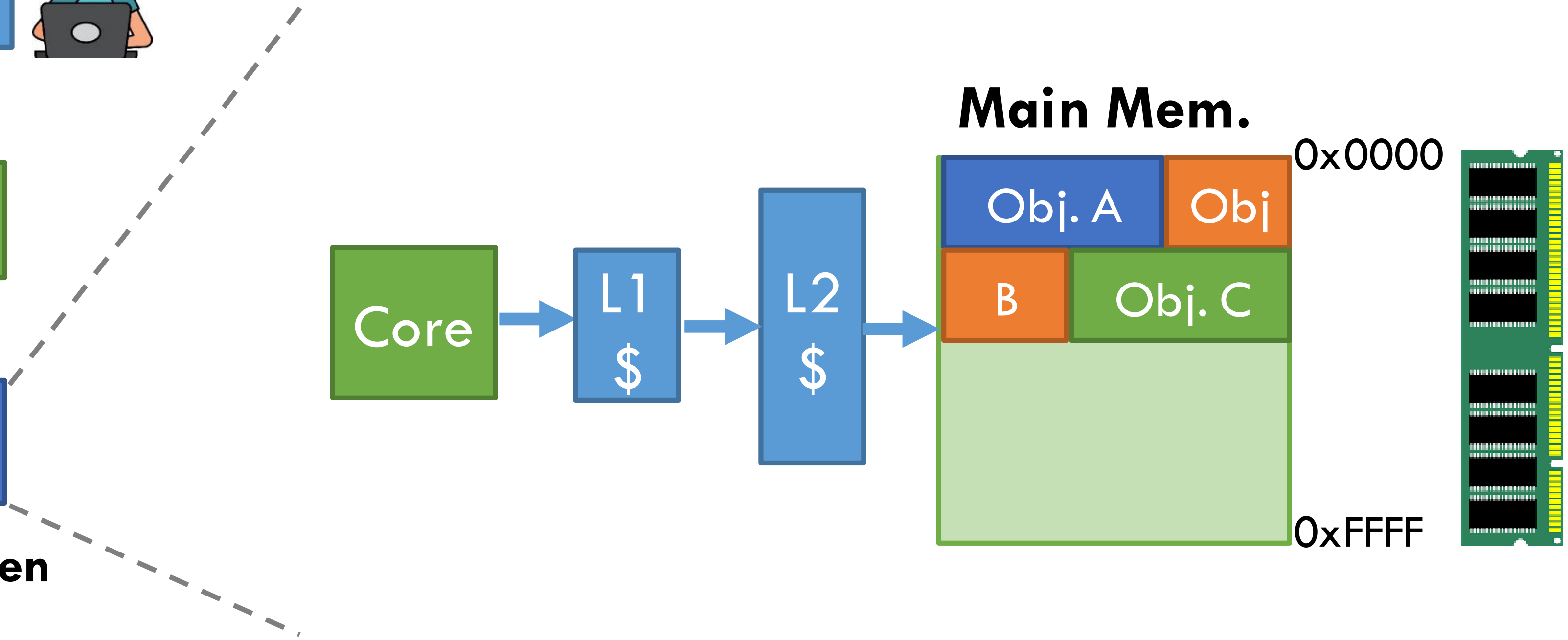
The inexpressive flat address space is inefficient



The inexpressive flat address space is inefficient

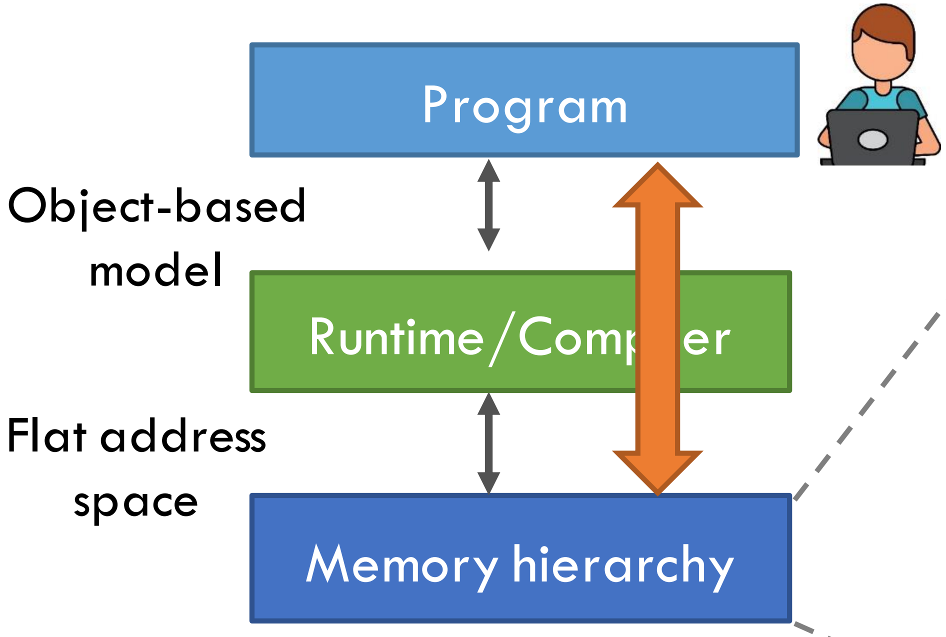


Semantic gap between programs and the memory hierarchy

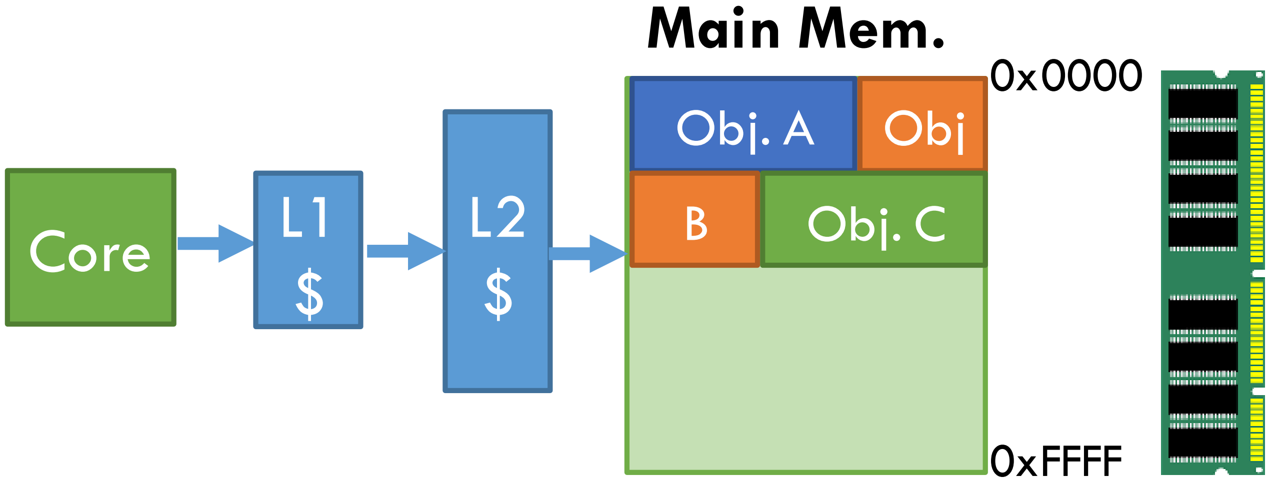


❌ Mismatch between objects and cache lines

The inexpressive flat address space is inefficient



Semantic gap between programs and the memory hierarchy



- ❌ Mismatch between objects and cache lines
- ❌ Costly associative lookups

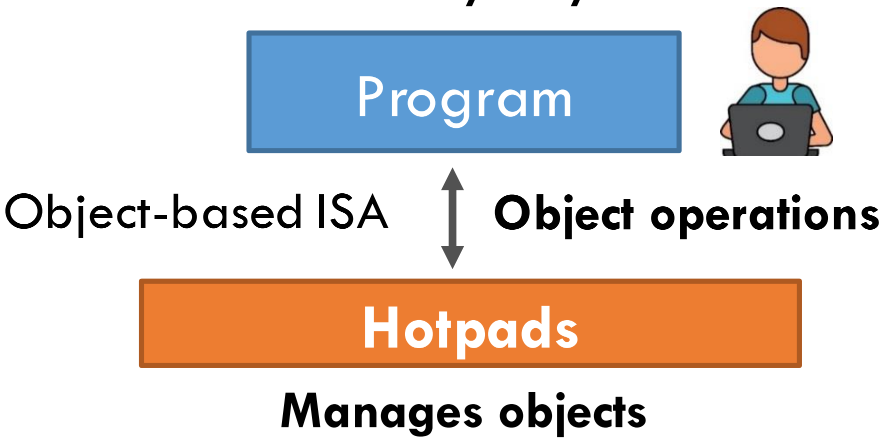
Hotpads: An object-based memory hierarchy

Hotpads: An object-based memory hierarchy

- A memory hierarchy designed from the ground up for *object-based programs*
  - Provides first-class support for objects and pointers in the ISA
  - Hides the memory layout from software and takes control over it

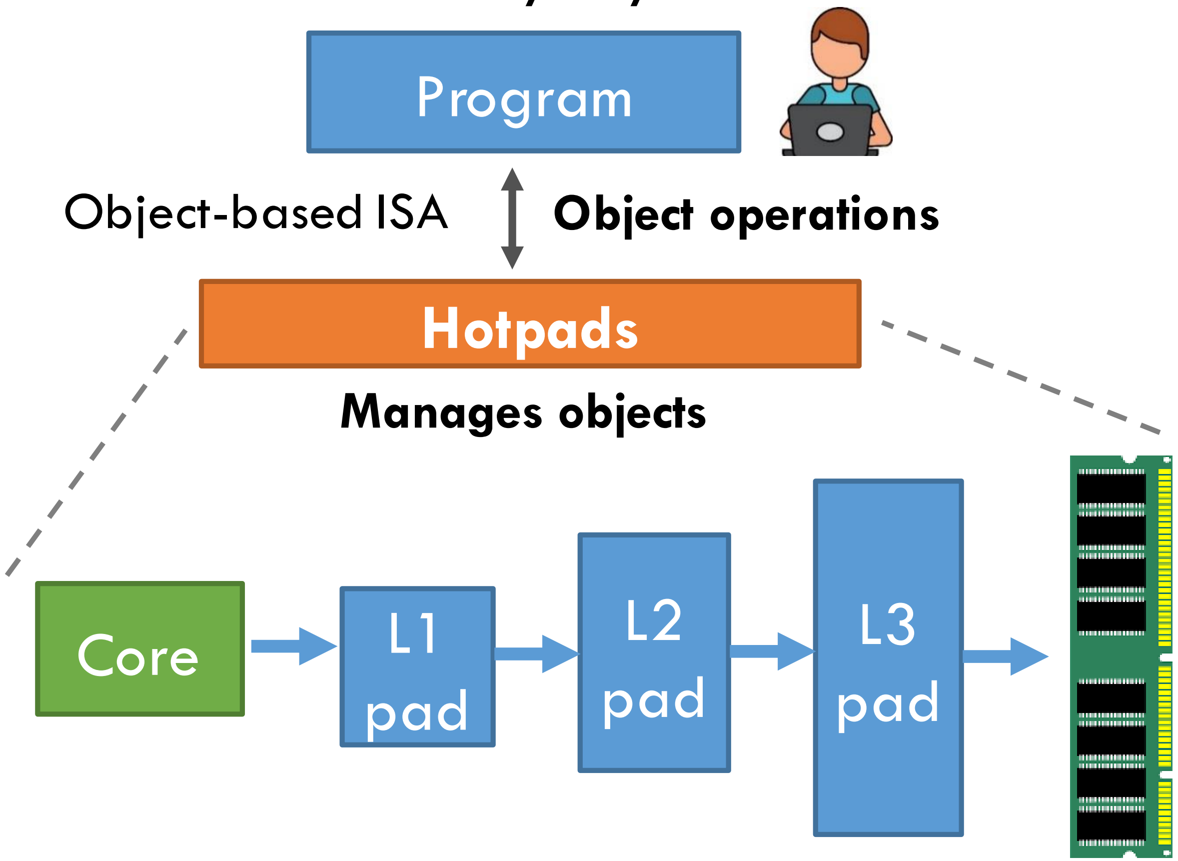
Hotpads: An object-based memory hierarchy

- A memory hierarchy designed from the ground up for *object-based programs*
  - Provides first-class support for objects and pointers in the ISA
  - Hides the memory layout from software and takes control over it



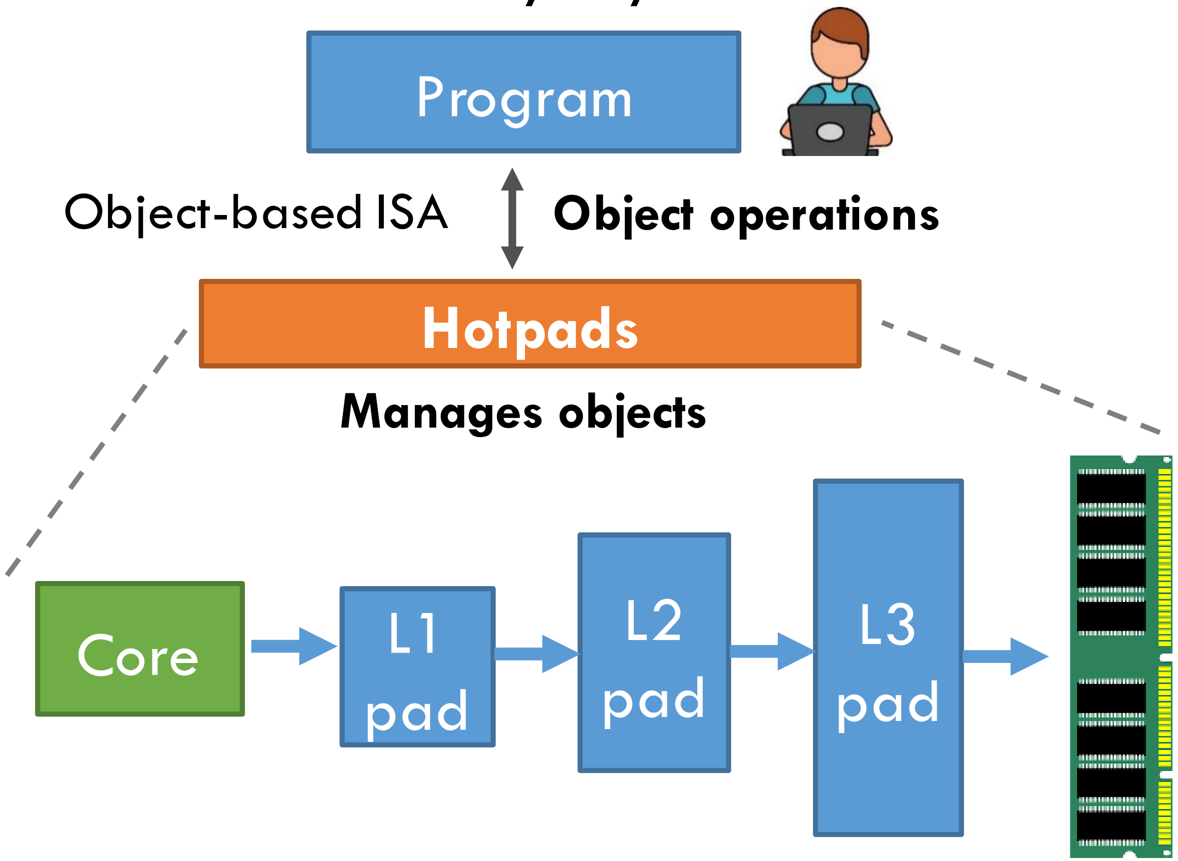
Hotpads: An object-based memory hierarchy

- A memory hierarchy designed from the ground up for *object-based programs*
  - Provides first-class support for objects and pointers in the ISA
  - Hides the memory layout from software and takes control over it



Hotpads: An object-based memory hierarchy

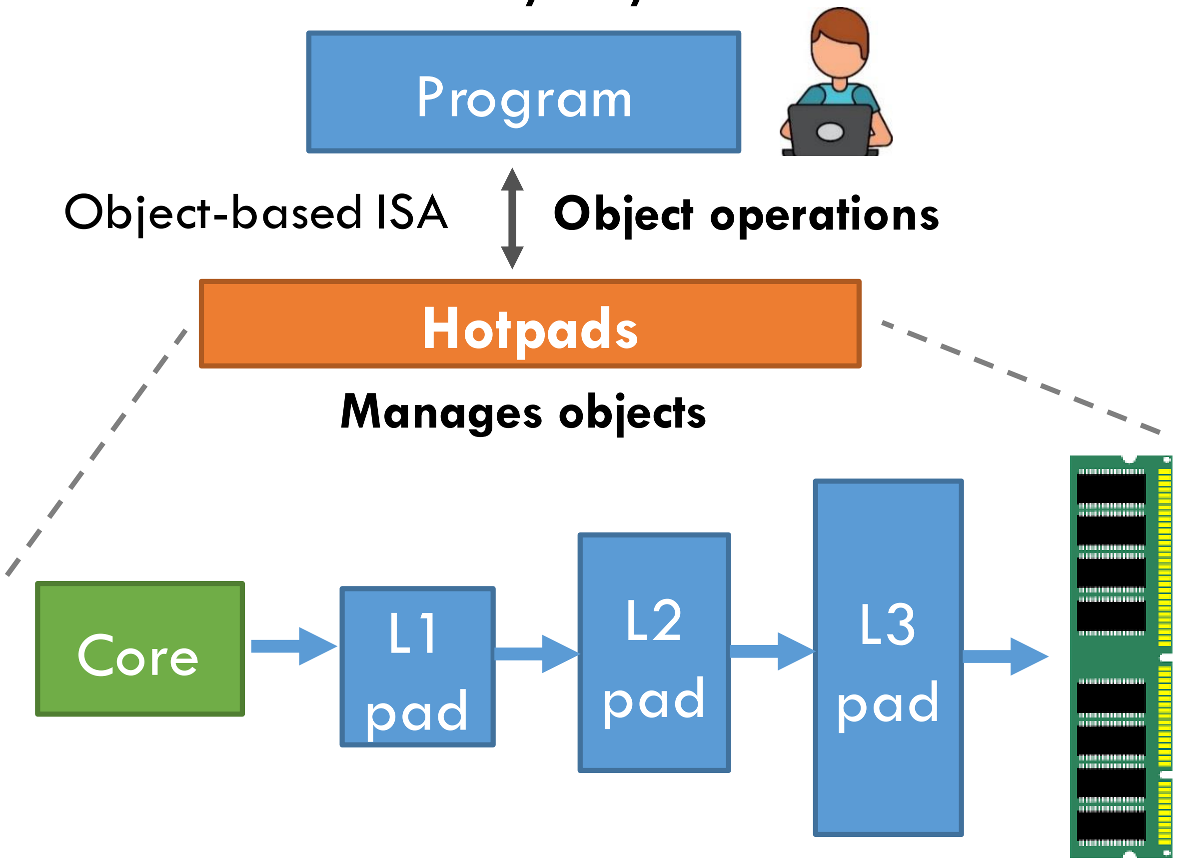
- A memory hierarchy designed from the ground up for *object-based programs*
  - Provides first-class support for objects and pointers in the ISA
  - Hides the memory layout from software and takes control over it



Hotpads manages objects instead of cache lines

Hotpads: An object-based memory hierarchy

- A memory hierarchy designed from the ground up for *object-based programs*
  - Provides first-class support for objects and pointers in the ISA
  - Hides the memory layout from software and takes control over it

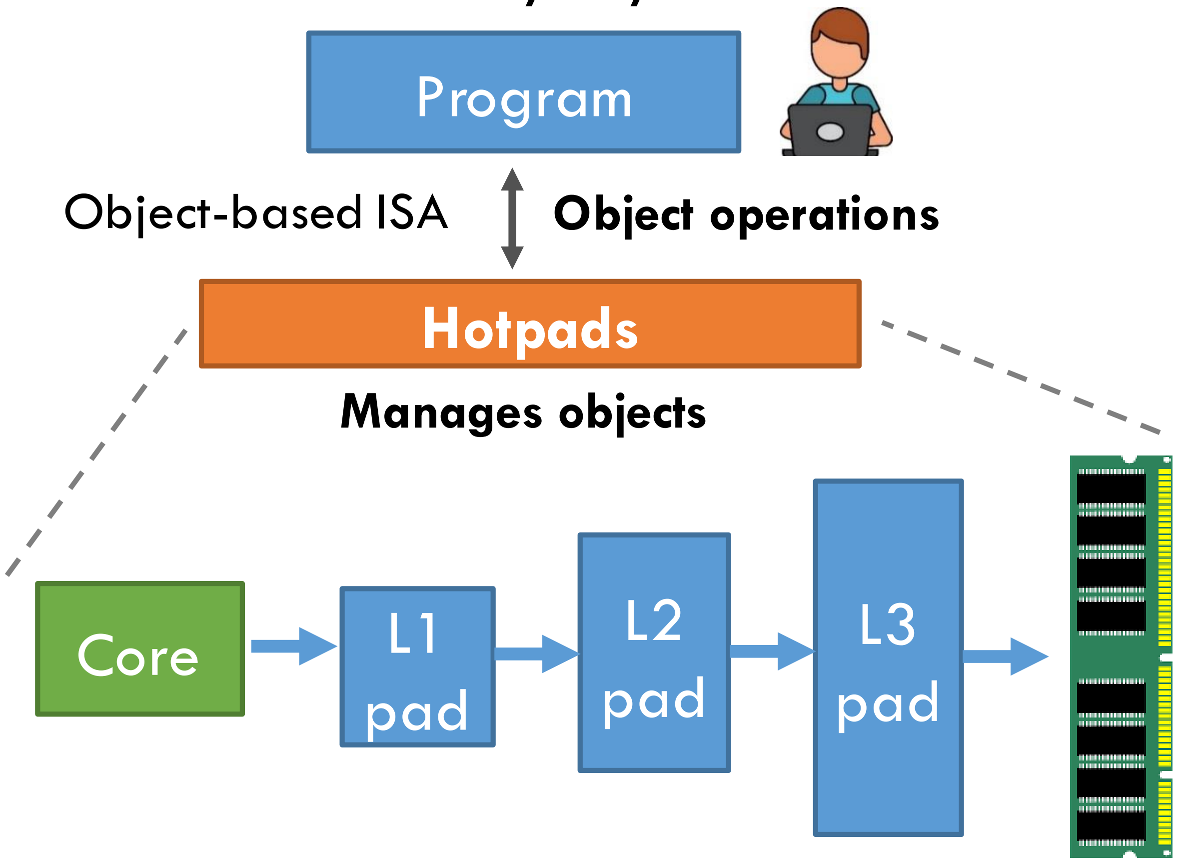


Hotpads manages objects instead of cache lines

Hotpads rewrites pointers to reduce associative lookups

Hotpads: An object-based memory hierarchy

- A memory hierarchy designed from the ground up for *object-based programs*
  - Provides first-class support for objects and pointers in the ISA
  - Hides the memory layout from software and takes control over it



Hotpads manages objects instead of cache lines

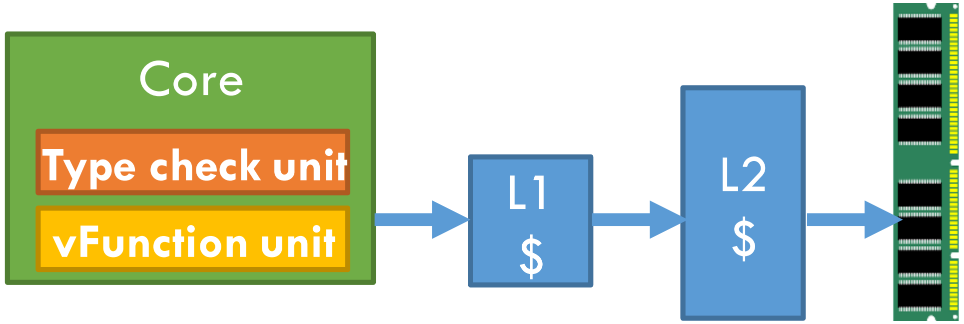
Hotpads rewrites pointers to reduce associative lookups

Hotpads provides architectural support for in-hierarchy object allocation and recycling

Prior architectural support for object-based programs

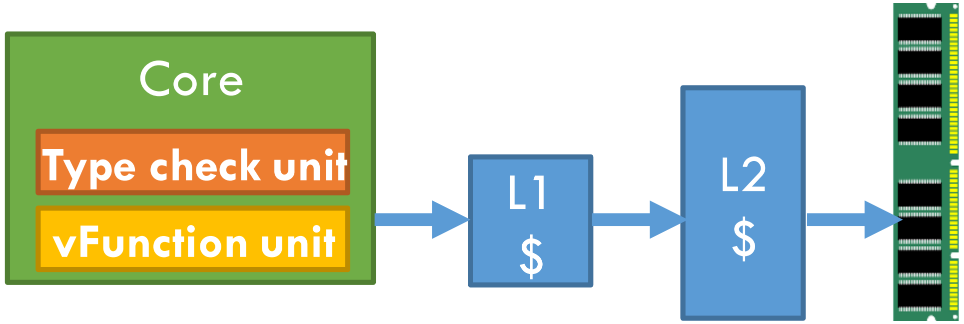
Prior architectural support for object-based programs

- Object-oriented/typed systems focus on **core microarchitecture design**
  - Accelerate virtual calls, object references and dynamic type checks

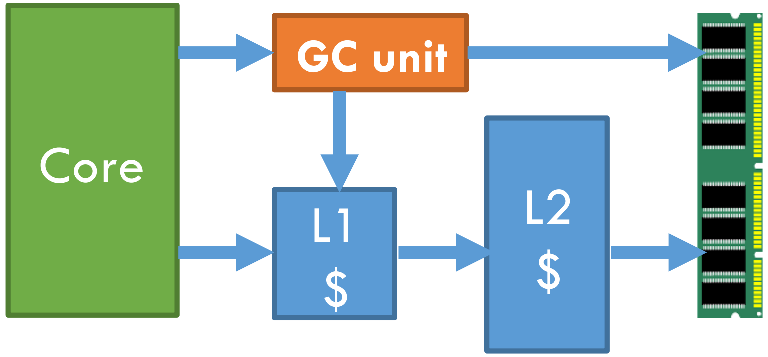


Prior architectural support for object-based programs

- Object-oriented/typed systems focus on **core microarchitecture design**
  - Accelerate virtual calls, object references and dynamic type checks

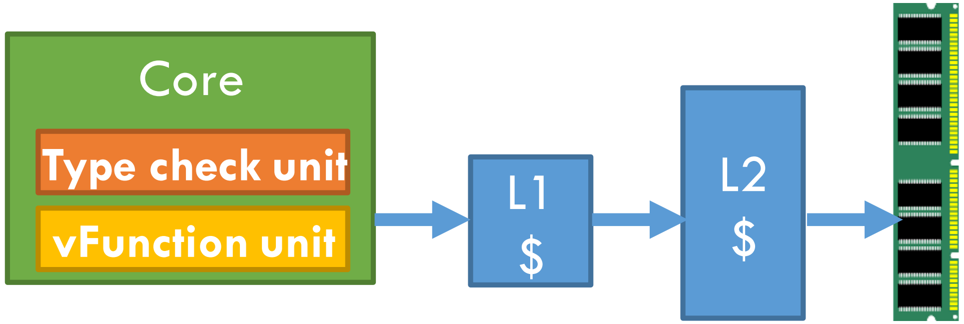


- Hardware accelerators for GC

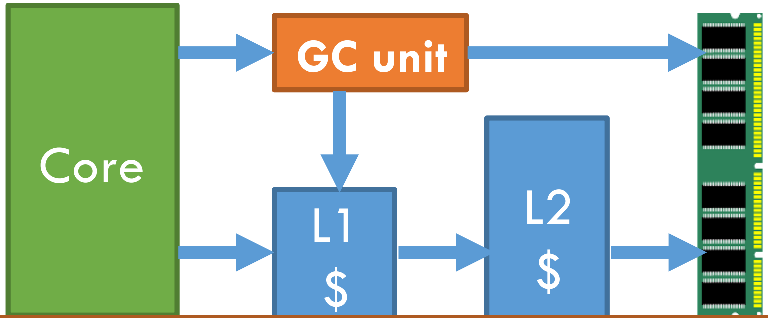


Prior architectural support for object-based programs

- Object-oriented/typed systems focus on **core microarchitecture design**
  - Accelerate virtual calls, object references and dynamic type checks



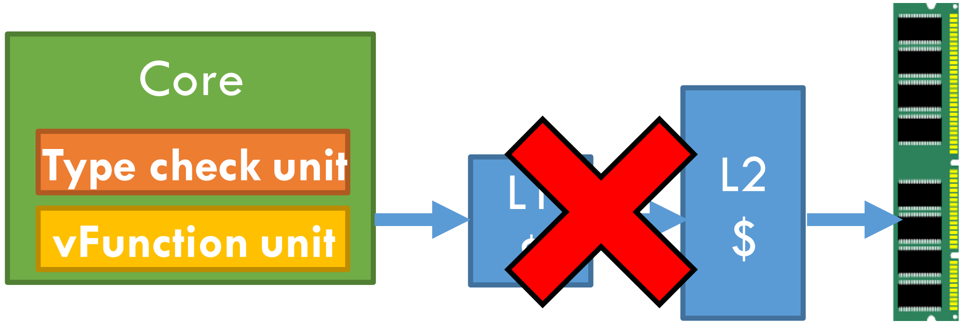
- Hardware accelerators for GC



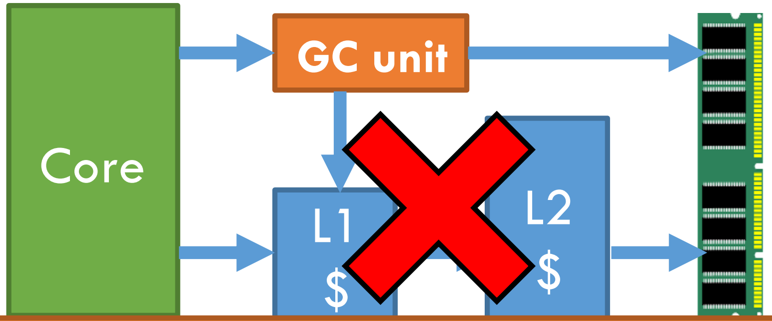
Prior work uses standard cache hierarchies

Prior architectural support for object-based programs

- Object-oriented/typed systems
  focus on **core microarchitecture design**
  - Accelerate virtual calls, object references and dynamic type checks



- Hardware accelerators for GC

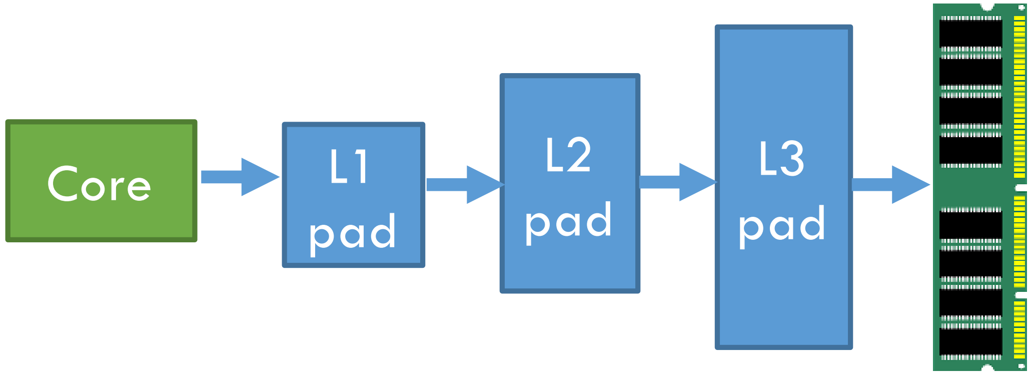


Prior work uses standard cache hierarchies

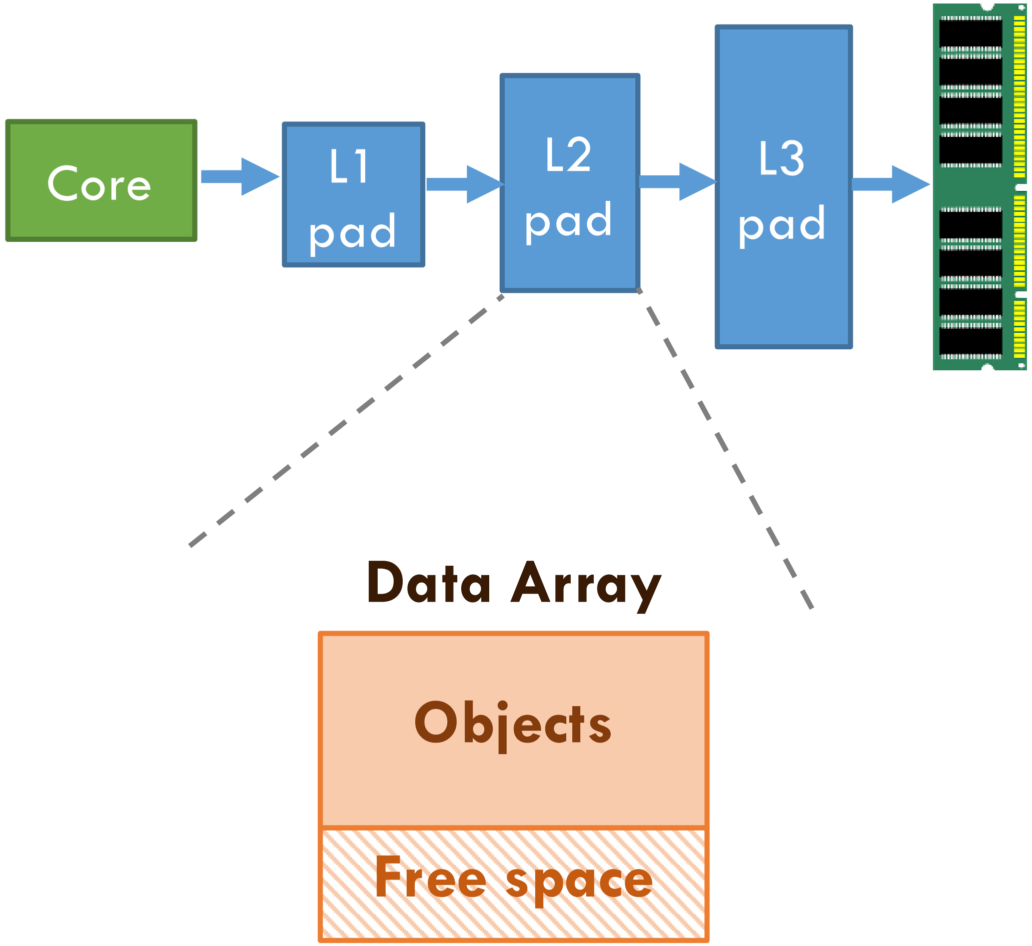
We focus on **redesigning the memory hierarchy**

Hotpads overview

Hotpads overview

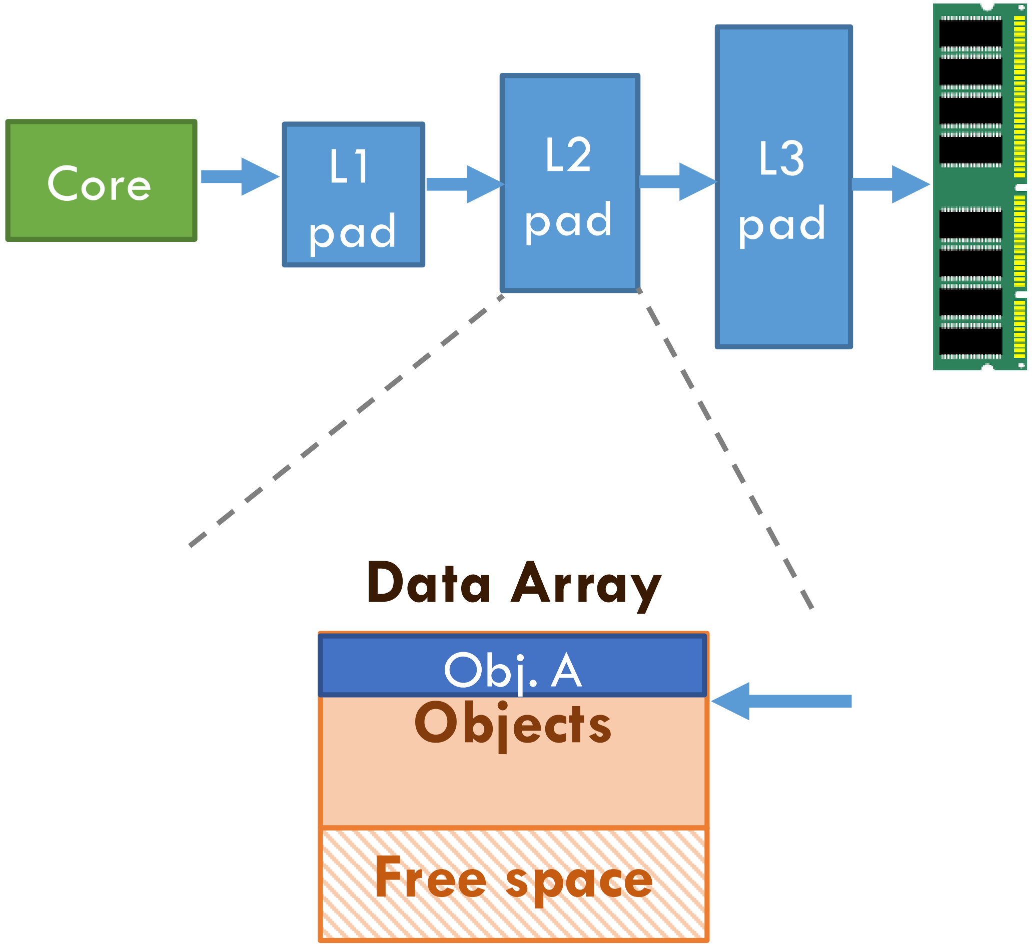


Hotpads overview



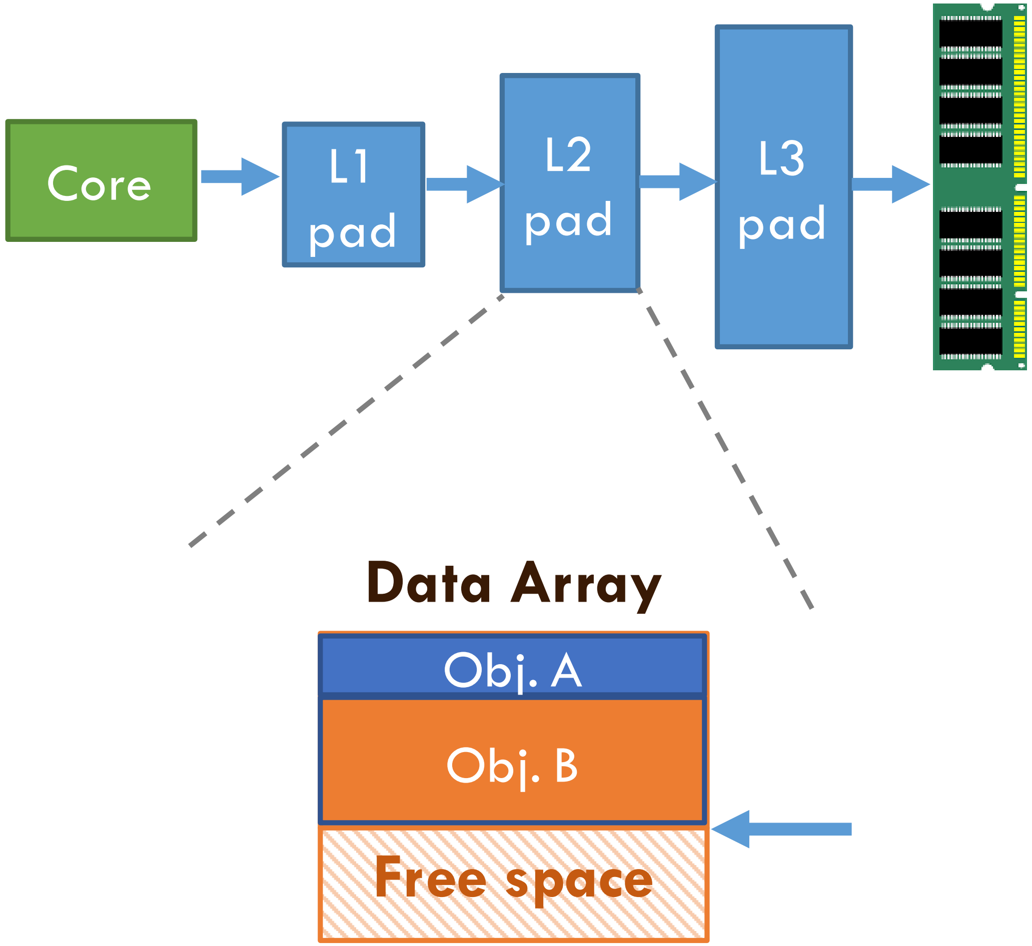
- Data array
  - Managed as a circular buffer using simple bump pointer allocation
  - Stores variable-sized objects compactly

Hotpads overview



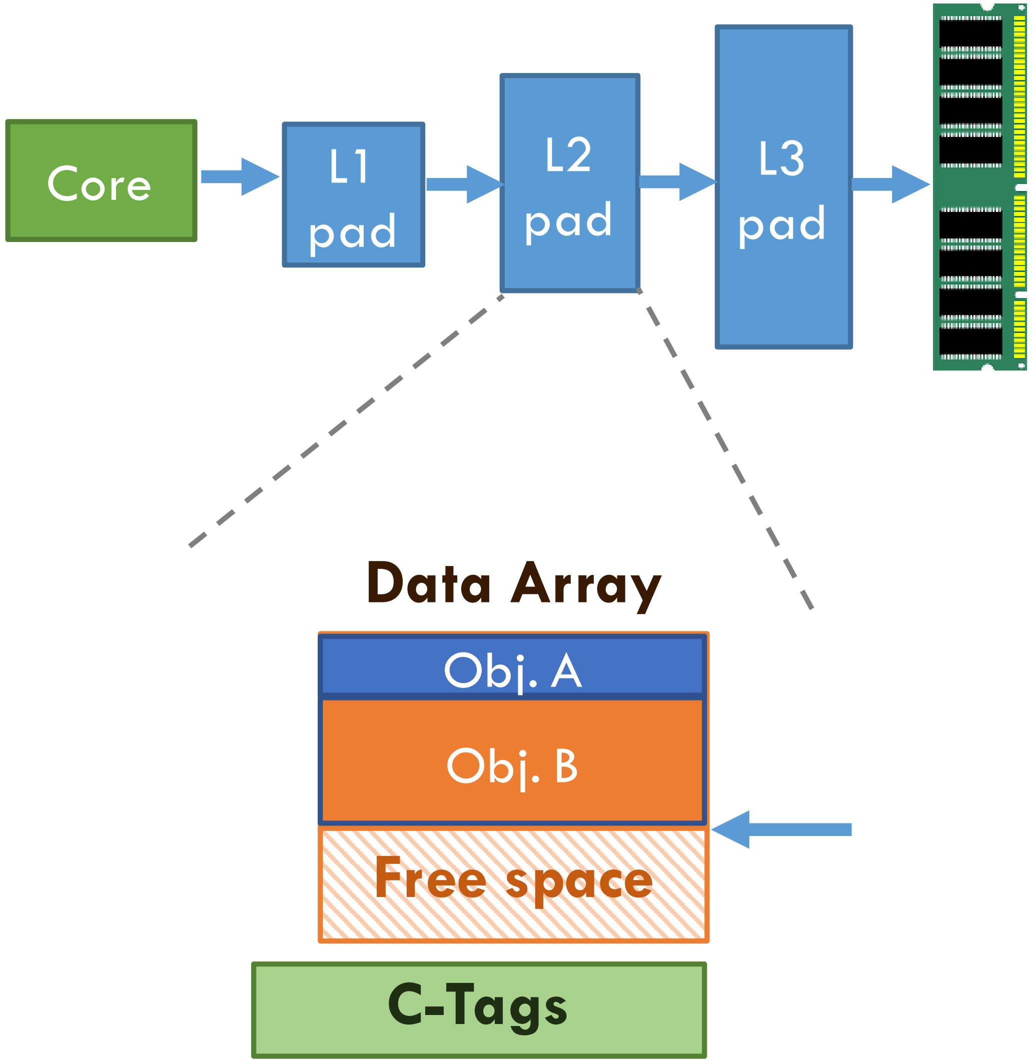
- Data array
  - Managed as a circular buffer using simple bump pointer allocation
  - Stores variable-sized objects compactly

Hotpads overview



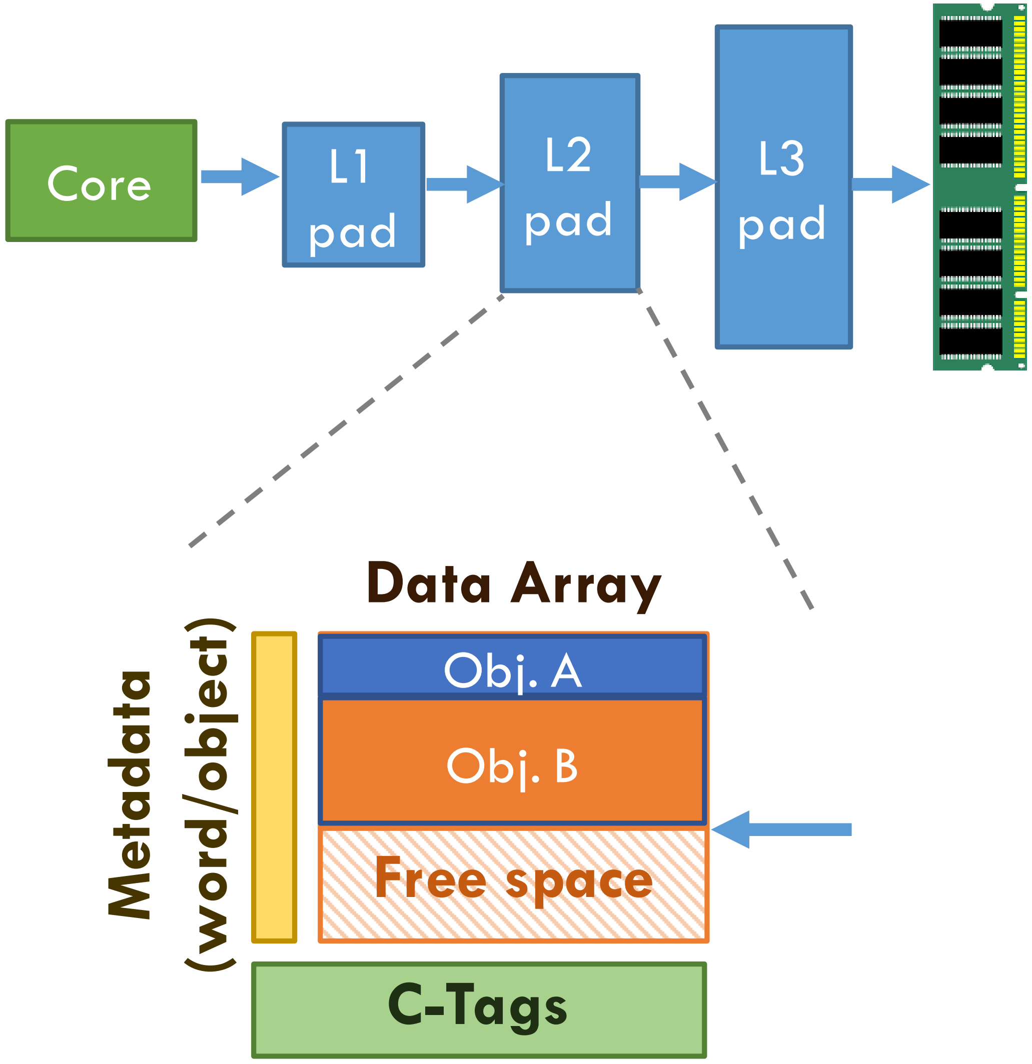
- Data array
  - Managed as a circular buffer using simple bump pointer allocation
  - Stores variable-sized objects compactly

Hotpads overview



- Data array
  - Managed as a circular buffer using simple bump pointer allocation
  - Stores variable-sized objects compactly
- C-Tags
  - Decoupled tag store used only for a fraction of accesses

Hotpads overview



- Data array
  - Managed as a circular buffer using simple bump pointer allocation
  - Stores variable-sized objects compactly
- C-Tags
  - Decoupled tag store used only for a fraction of accesses
- Metadata
  - Pointer? valid? dirty? recently-used?

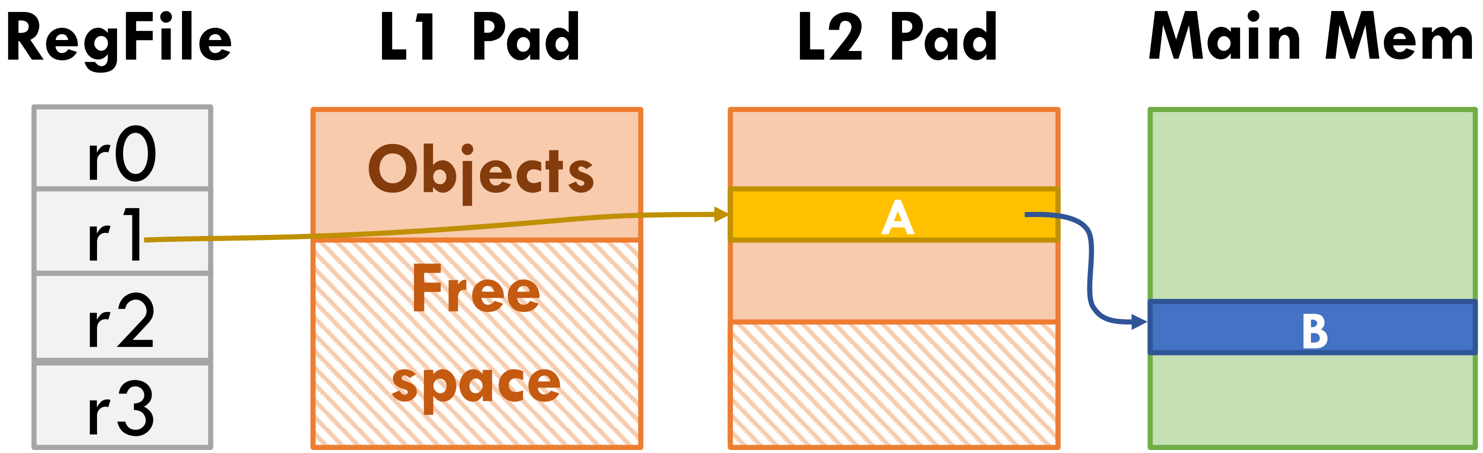
Hotpads example

Hotpads example

```
class Node {
    int value;
    Node next;
}
```

Hotpads example

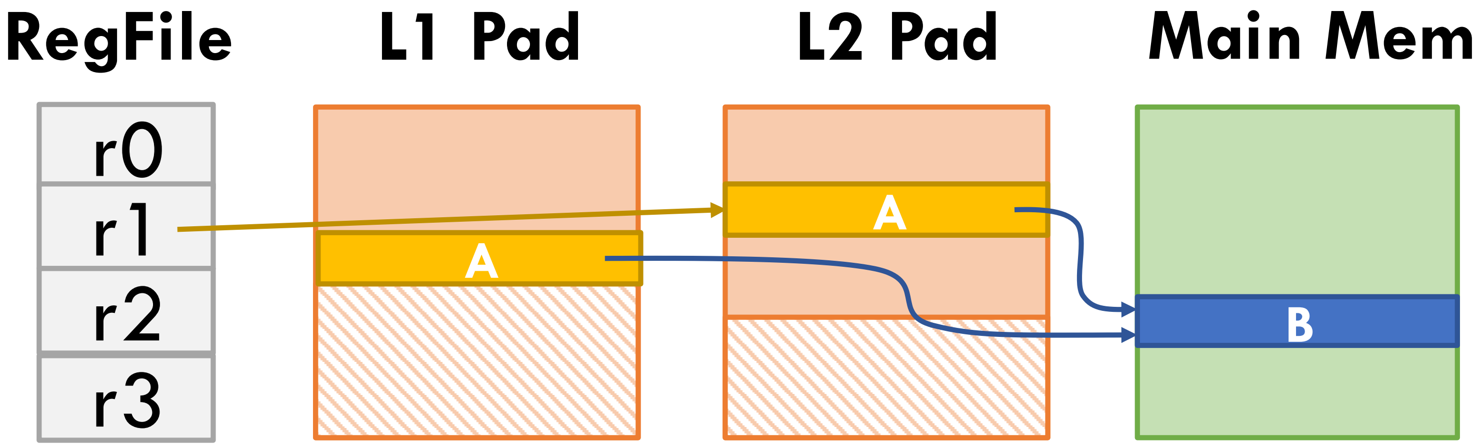
```
class Node {
    int value;
    Node next;
}
```



Initial state.

Hotpads moves object implicitly

```
class Node {
    int value;
    Node next;
}
```



Program code:

```
int v = A.value;
```

Hotpads instructions:

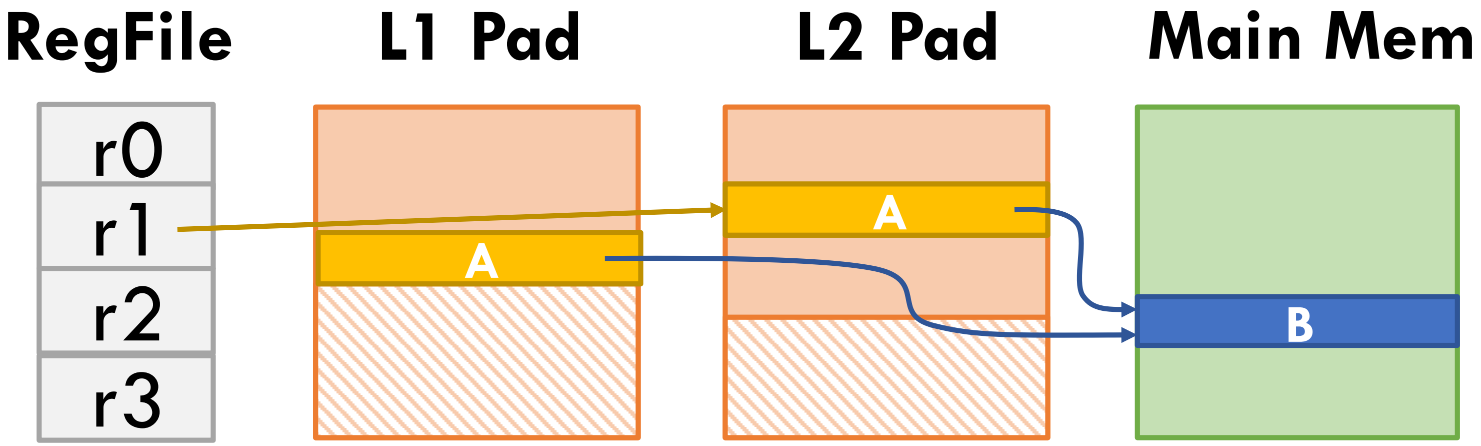
```
ld r0, (r1).value
```

Core issues access to A.

A is copied into L1 pad.

Hotpads moves object implicitly

```
class Node {
    int value;
    Node next;
}
```



Program code:

```
int v = A.value;
```

Hotpads instructions:

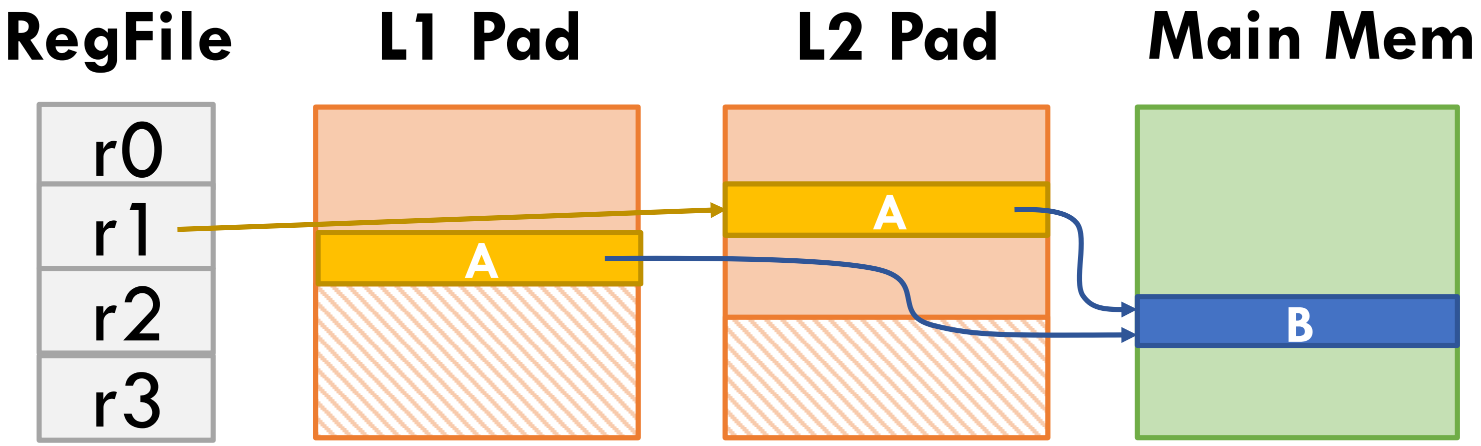
```
ld r0, (r1).value
```

Core issues access to A.
A is copied into L1 pad.

- All loads/stores follow a single addressing mode: Base+offset

Hotpads moves object implicitly

```
class Node {
    int value;
    Node next;
}
```



Program code:

```
int v = A.value;
```

Hotpads instructions:

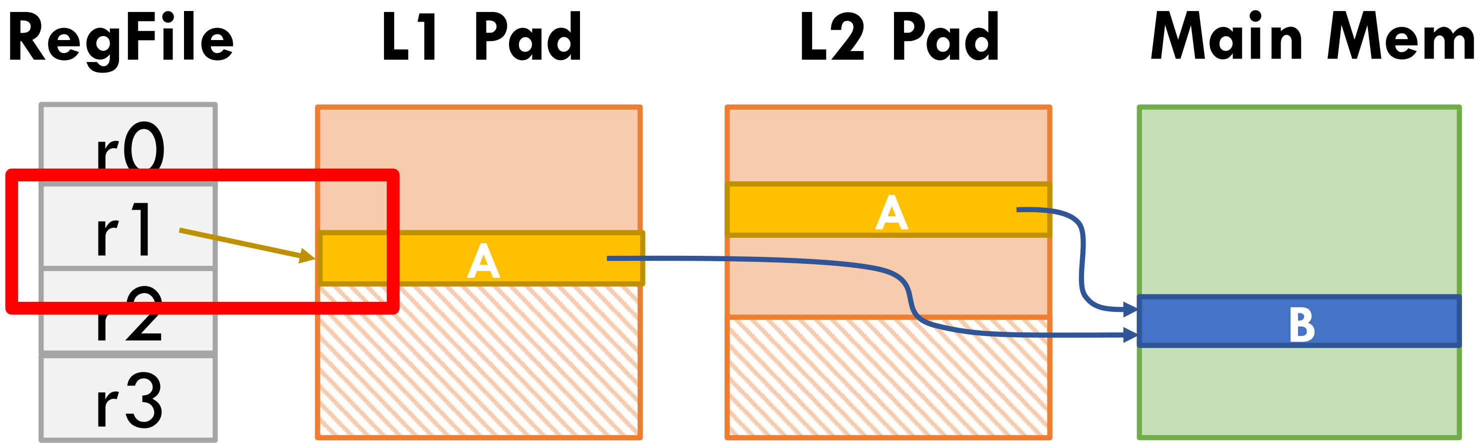
```
ld r0, (r1).value
```

Core issues access to A.
A is copied into L1 pad.

- All loads/stores follow a single addressing mode: Base+offset
- Bump pointer allocation stores A compactly after other objects

Hotpads rewrites pointers to avoid associative lookups

```
class Node {
    int value;
    Node next;
}
```



Program code:

```
int v = A.value;
```

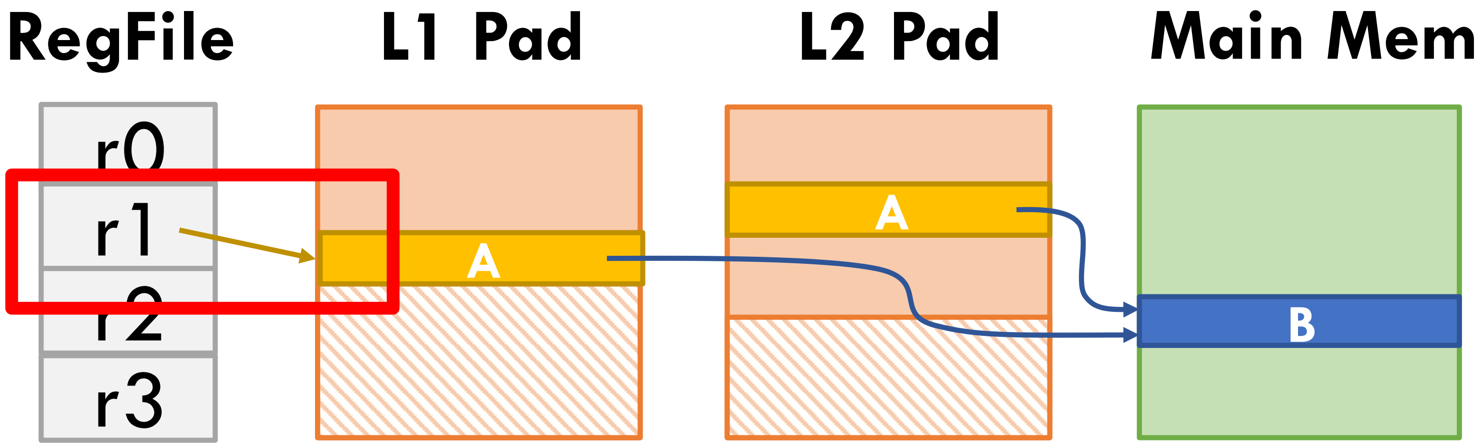
Hotpads instructions:

```
ld r0, (r1).value
```

Core issues access to A.
A is copied into L1 pad.
r1 is rewritten to A's L1 pad address.

Hotpads rewrites pointers to avoid associative lookups

```
class Node {
    int value;
    Node next;
}
```



Program code:

```
int v = A.value;
```

Hotpads instructions:

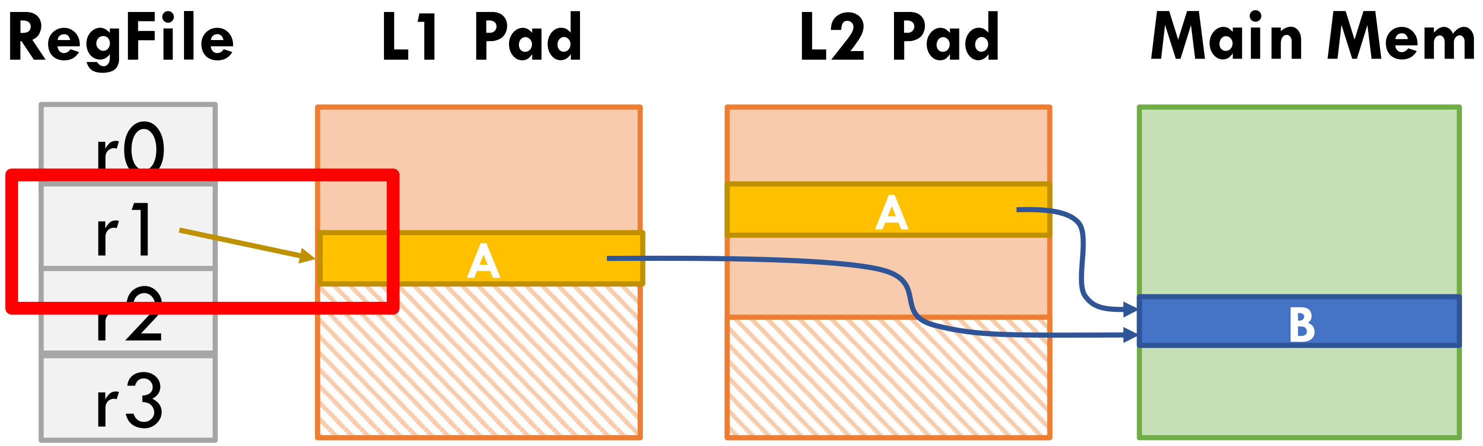
```
ld r0, (r1).value
```

Core issues access to A.
A is copied into L1 pad.
r1 is rewritten to A's L1 pad address.

- Subsequent dereferences of r1 access A's L1 copy directly, **without associative lookups (like a scratchpad)**

Hotpads rewrites pointers to avoid associative lookups

```
class Node {
    int value;
    Node next;
}
```



Program code:
```
int v = A.value;
```

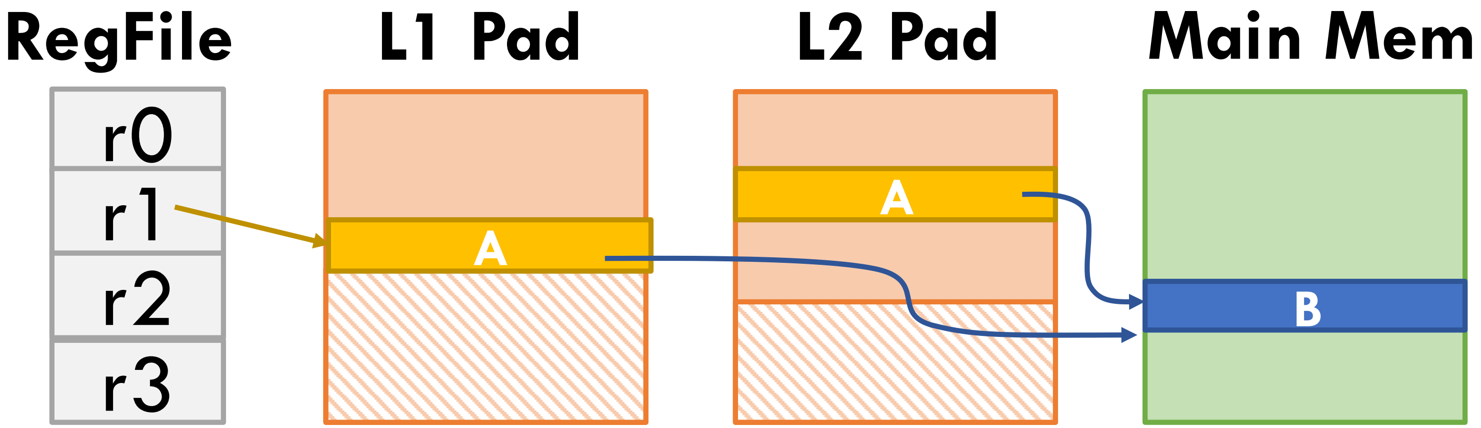
Hotpads instructions:
```
ld r0, (r1).value
```

Core issues access to A.
A is copied into L1 pad.
r1 is rewritten to A's L1 pad address.

- Subsequent dereferences of r1 access A's L1 copy directly, **without associative lookups (like a scratchpad)**
- Hotpads rewrites pointers safely because it hides the memory layout from software

Pointer rewriting applies to L1 pad data as well

```
class Node {
    int value;
    Node next;
}
```



B copied into L1.

A's pointer is rewritten.

Program code:

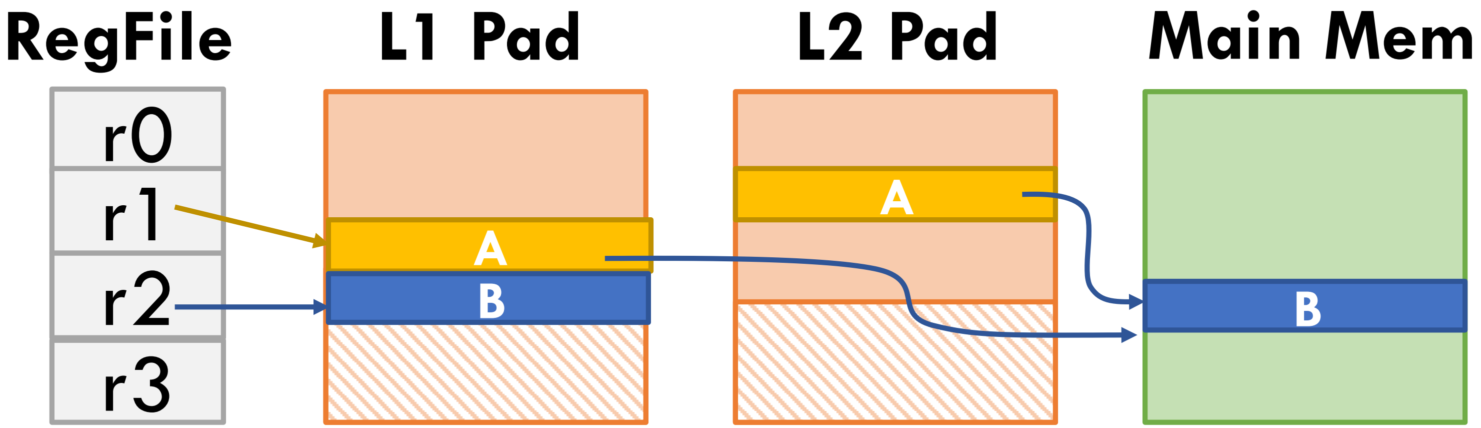
```
v = A.next.value;
```

Hotpads instructions:

```
derefptr r2, (r1).next
ld r3, (r2).value
```

Pointer rewriting applies to L1 pad data as well

```
class Node {
    int value;
    Node next;
}
```



B copied into L1.
A's pointer is rewritten.

Program code:

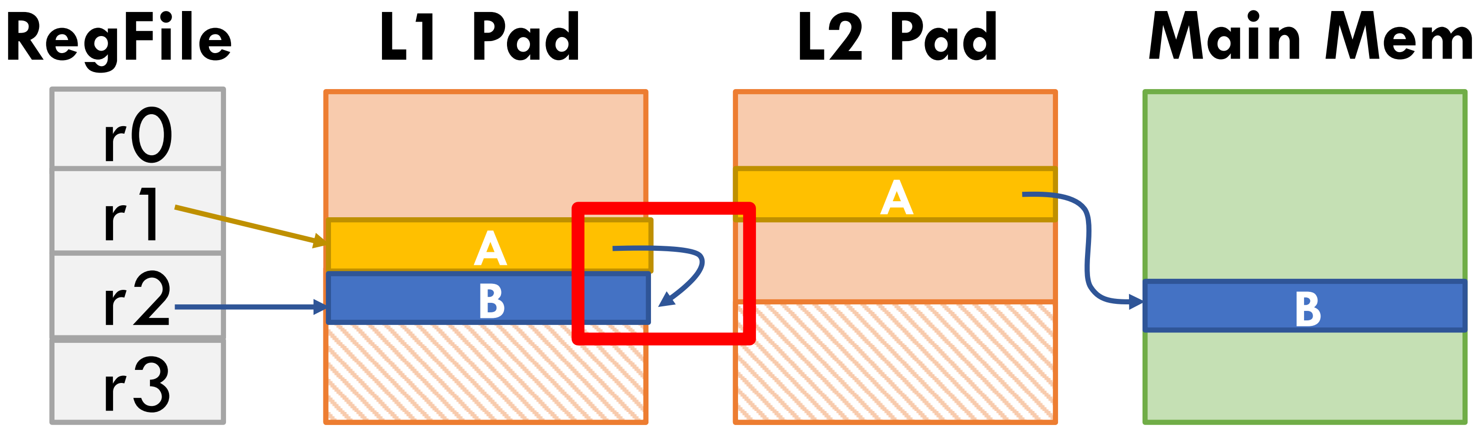
```
v = A.next.value;
```

Hotpads instructions:

```
derefptr r2, (r1).next
ld r3, (r2).value
```

Pointer rewriting applies to L1 pad data as well

```
class Node {
    int value;
    Node next;
}
```



B copied into L1.
A's pointer is rewritten.

Program code:

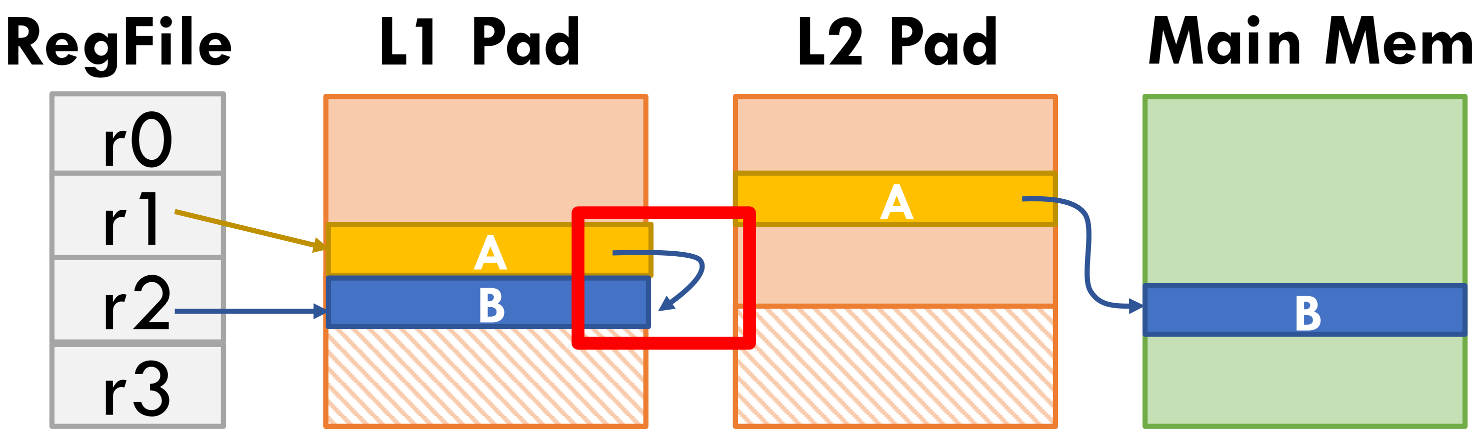
```
v = A.next.value;
```

Hotpads instructions:

```
derefptr r2, (r1).next
ld r3, (r2).value
```

Pointer rewriting applies to L1 pad data as well

```
class Node {
    int value;
    Node next;
}
```



Program code:

```
v = A.next.value;
```

Hotpads instructions:

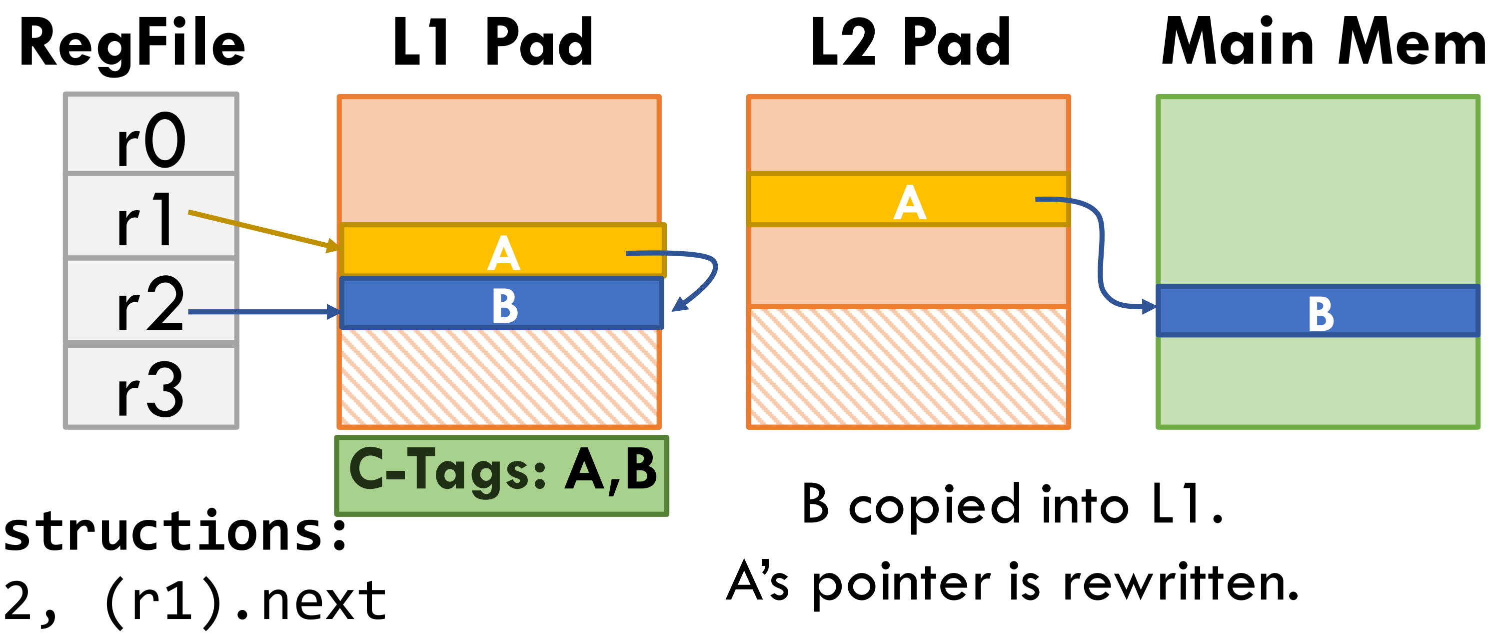
```
derefptr r2, (r1).next
ld r3, (r2).value
```

B copied into L1.
A's pointer is rewritten.

- Subsequent dereferences of A.next access the L1 copy of B directly, **without associative lookups**

Pointer rewriting applies to L1 pad data as well

```
class Node {
    int value;
    Node next;
}
```



Program code:

```
v = A.next.value;
```

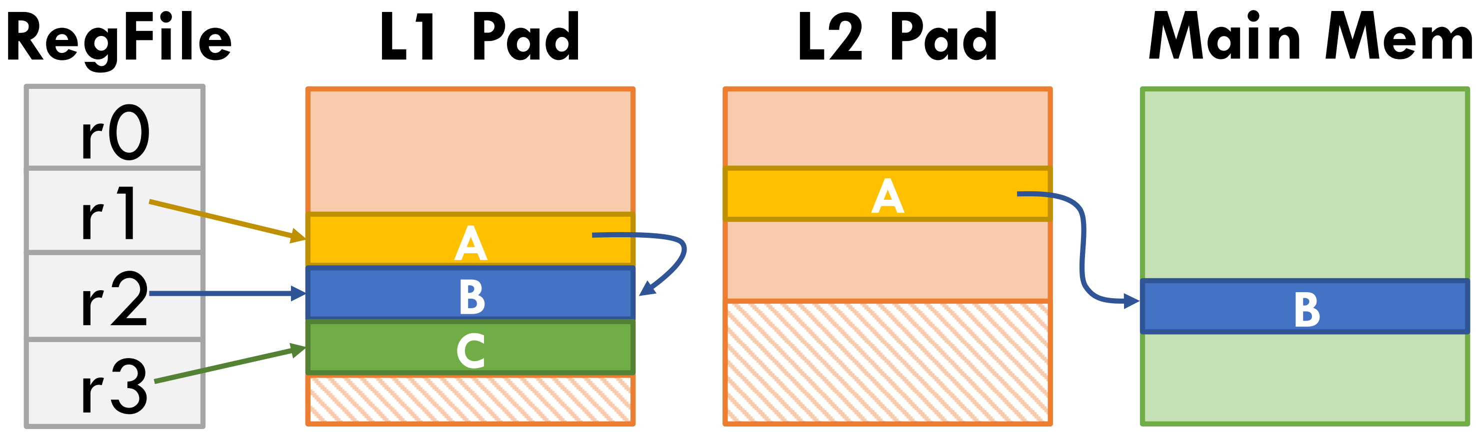
Hotpads instructions:

```
derefptr r2, (r1).next
ld r3, (r2).value
```

- Subsequent dereferences of A.next access the L1 copy of B directly, **without associative lookups**
- C-tags let dereferencing other pointers of A and B find their L1 copies

Hotpads supports in-hierarchy object allocation

```
class Node {
    int value;
    Node next;
}
```

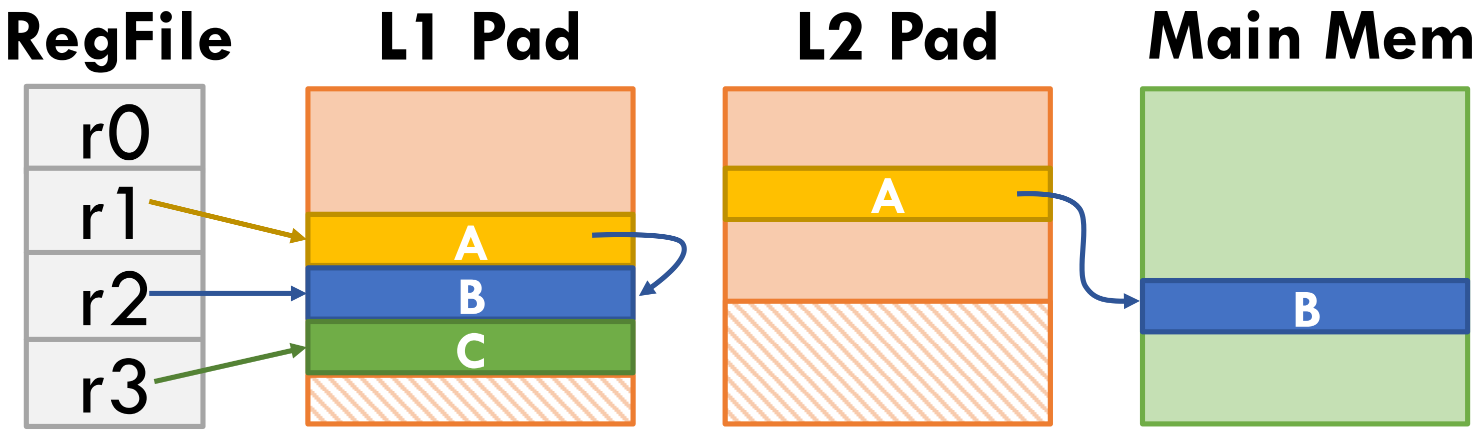


Core allocates new object C.

```
Program code:          Hotpads instructions:
Node C = new Node(); alloc r3, type=Node
```

Hotpads supports in-hierarchy object allocation

```
class Node {
    int value;
    Node next;
}
```



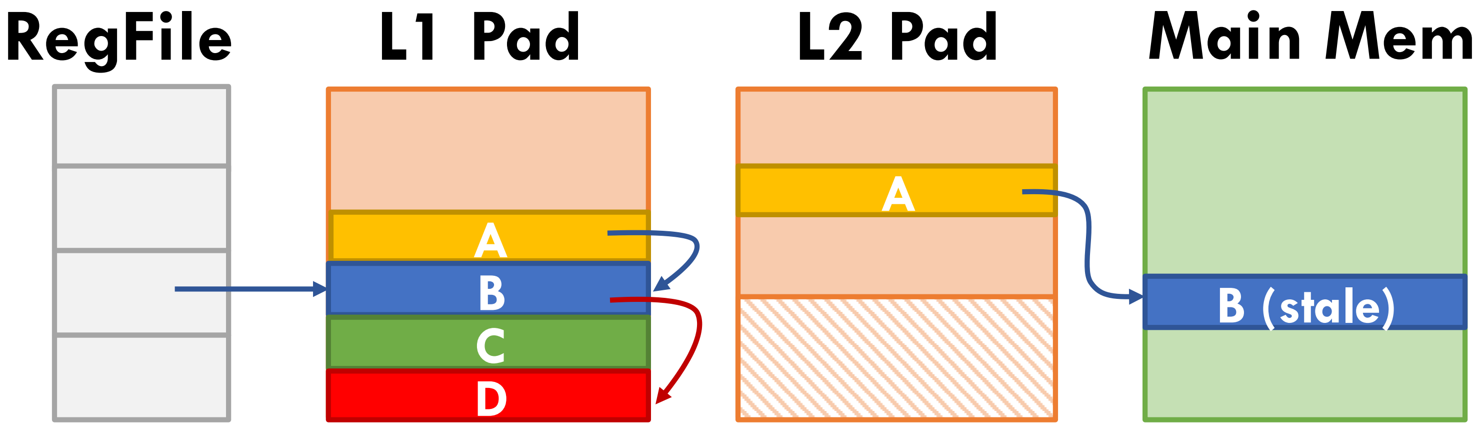
Core allocates new object C.

Program code:
`Node C = new Node();`

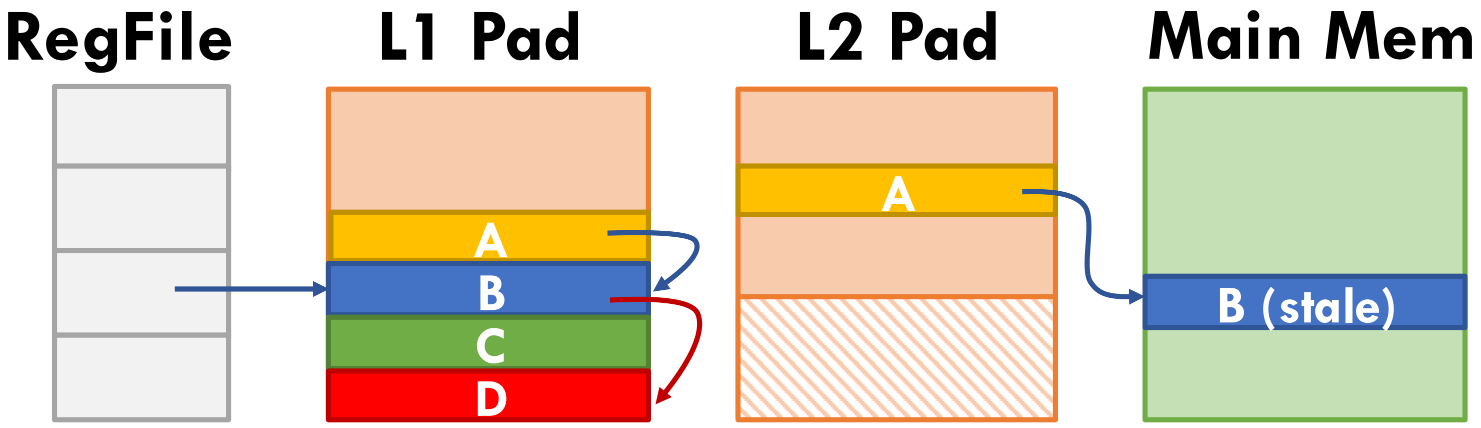
Hotpads instructions:
`alloc r3, type=Node`

- In-hierarchy allocation reduces data movement and requires no backing storage in main memory or larger pads

Hotpads unifies garbage collection and object evictions

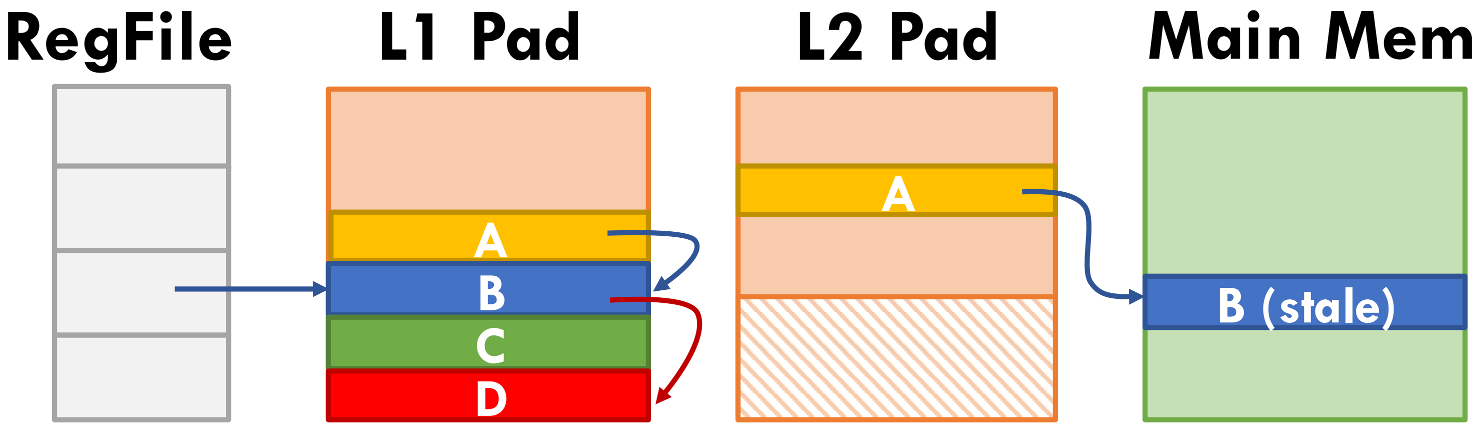


Hotpads unifies garbage collection and object evictions



L1 pad is now full

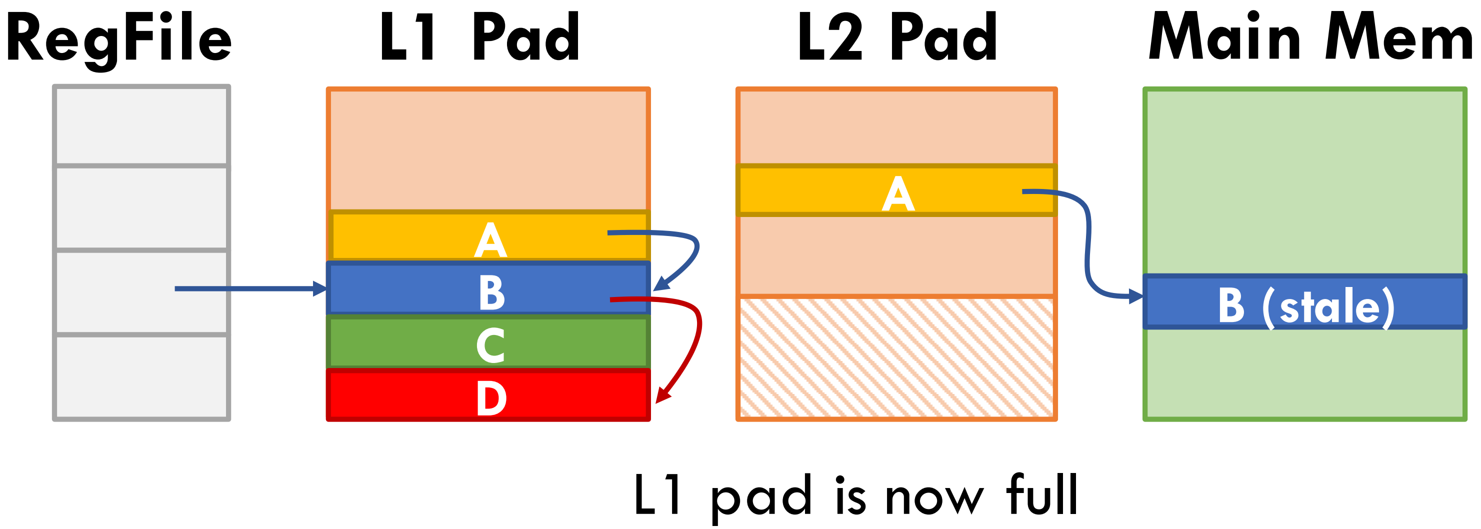
Hotpads unifies garbage collection and object evictions



L1 pad is now full

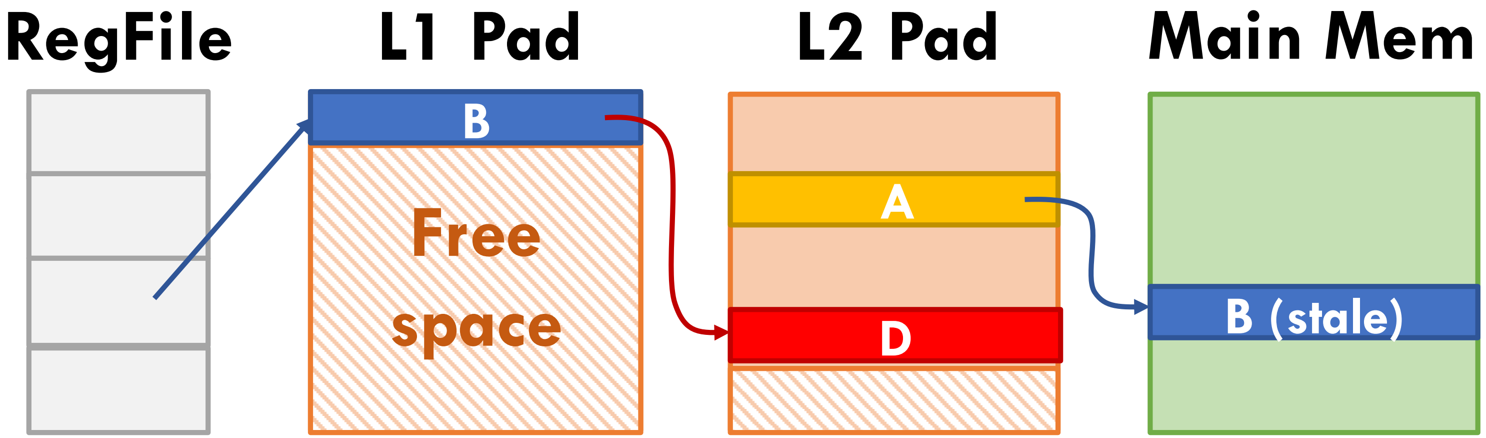
- When a pad fills up, it triggers a collection-eviction (CE) to free space
  - Discards dead objects
  - Evicts live, non-recently used objects to the next level in bulk

Hotpads unifies garbage collection and object evictions



- When a pad fills up, it triggers a collection-eviction (CE) to free space
  - Discards dead objects
  - Evicts live, non-recently used objects to the next level in bulk
- C is dead (unreferenced). Other objects are live. Only B is recently used.

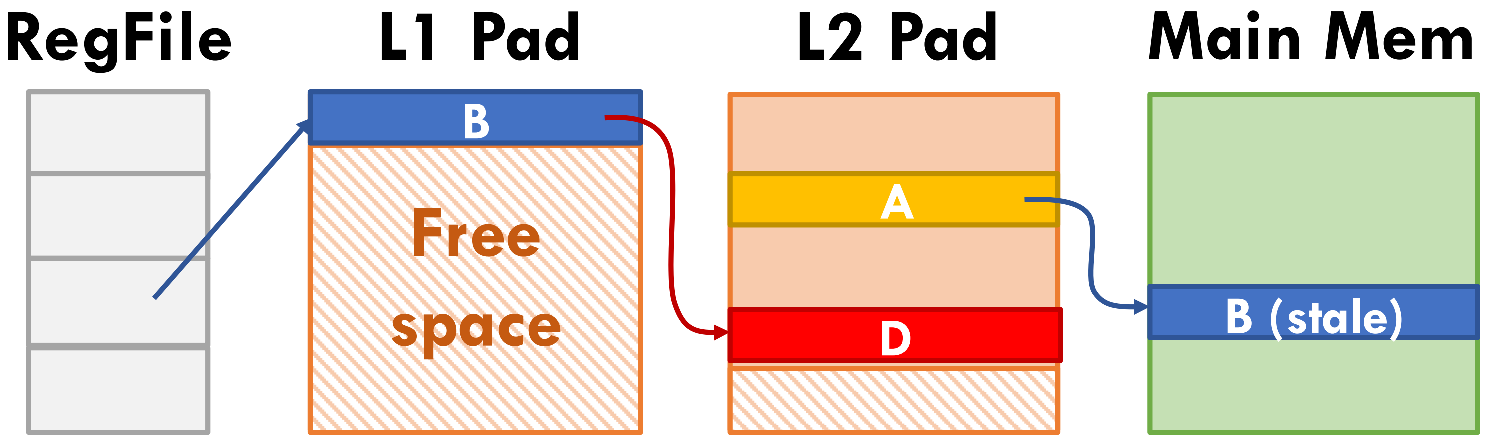
Hotpads unifies garbage collection and object evictions



L1 collection-eviction (CE) collects dead C and evicts live A & D to L2. It leaves a large contiguous chunk of free space

Hotpads unifies garbage collection and object evictions

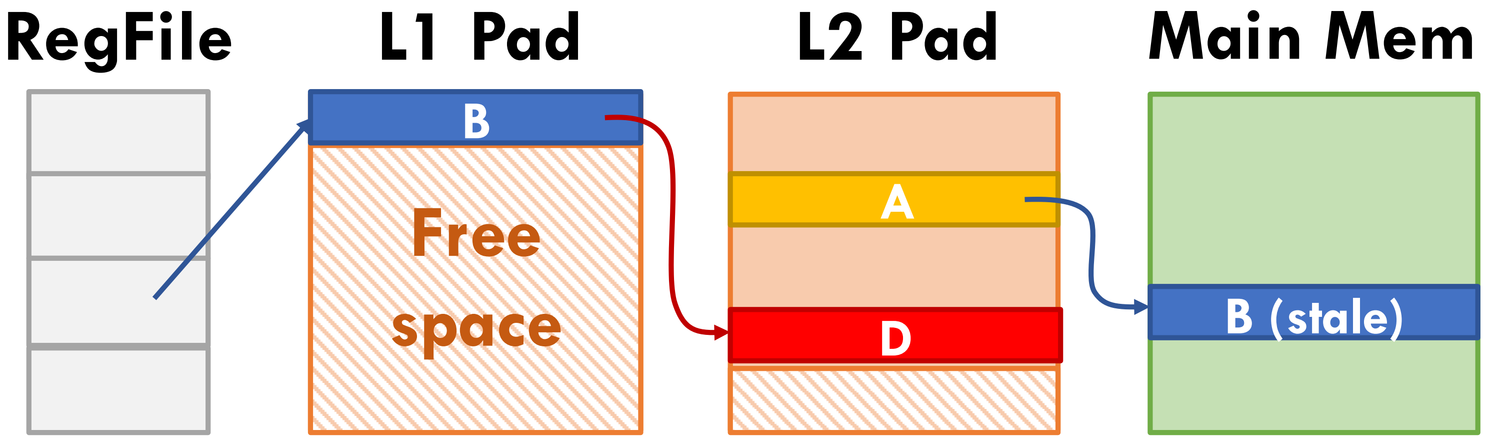
- CEs happen concurrently with program execution and are hierarchical



L1 collection-eviction (CE) collects dead C and evicts live A & D to L2. It leaves a large contiguous chunk of free space

Hotpads unifies garbage collection and object evictions

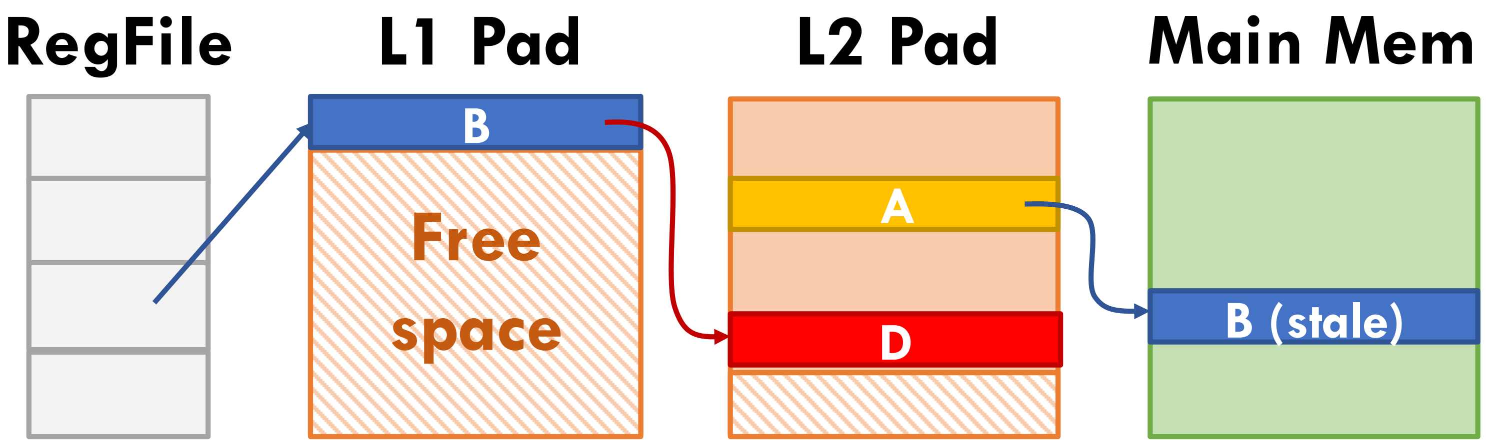
- CEs happen concurrently with program execution and are hierarchical
- Each pad can perform a CE independently from larger, higher-level pads → Makes CE cost proportional to pad size



L1 collection-eviction (CE) collects dead C and evicts live A & D to L2. It leaves a large contiguous chunk of free space

Hotpads unifies garbage collection and object evictions

- CEs happen concurrently with program execution and are hierarchical
- Each pad can perform a CE independently from larger, higher-level pads → Makes CE cost proportional to pad size

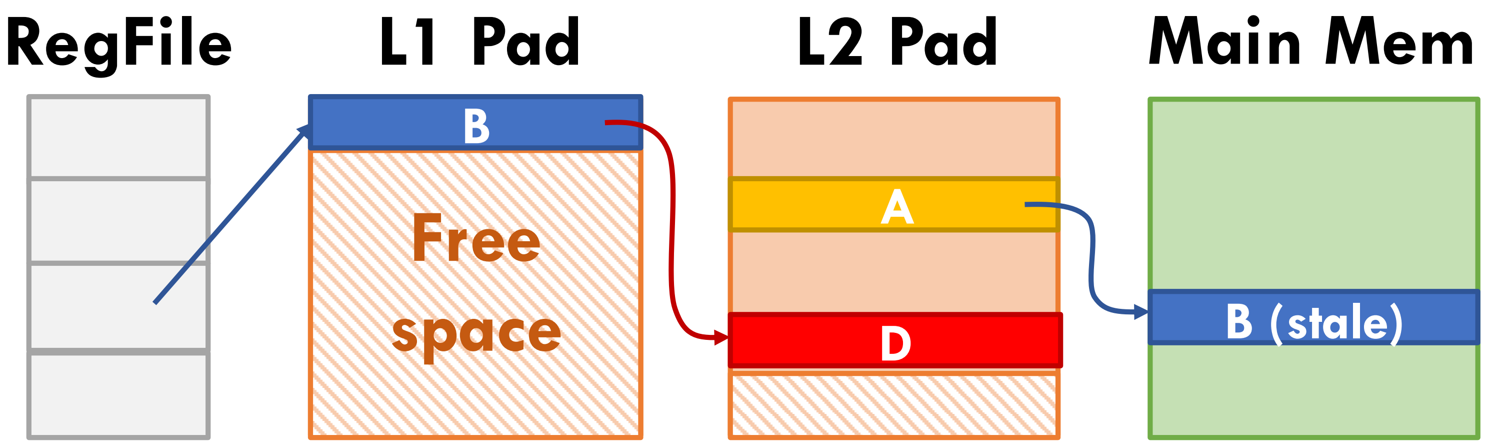


L1 collection-eviction (CE) collects dead C and evicts live A & D to L2. It leaves a large contiguous chunk of free space

Invariant: Objects at a particular level may only point to objects at the same or larger levels.

Hotpads unifies garbage collection and object evictions

- CEs happen concurrently with program execution and are hierarchical
- Each pad can perform a CE independently from larger, higher-level pads → Makes CE cost proportional to pad size



L1 collection-eviction (CE) collects dead C and evicts live A & D to L2. It leaves a large contiguous chunk of free space

Invariant: Objects at a particular level may only point to objects at the same or larger levels.

Result: No need to check the L2 pad when performing a collection-eviction in the L1 pad.

Collection-evictions reduce data movement

Collection-evictions reduce data movement

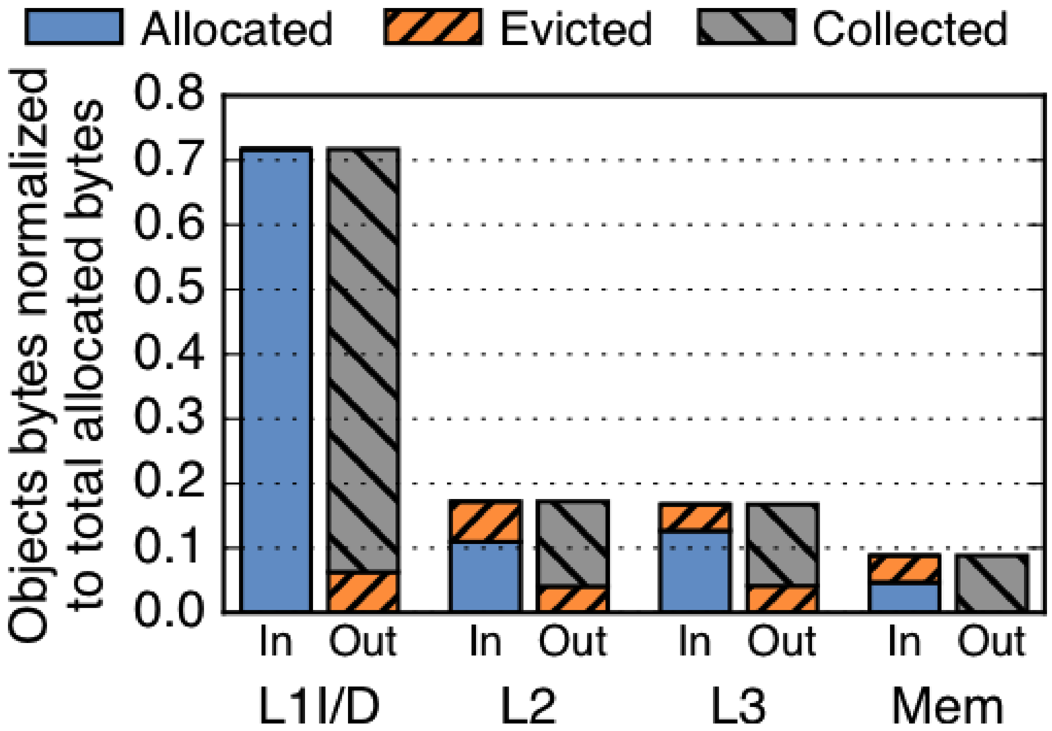
- Hotpads unifies the locality principle and the generational hypothesis

Collection-evictions reduce data movement

- Hotpads unifies the locality principle and the generational hypothesis
- Hotpads acts like a super-generational collector
  - Accesses to short-lived objects are cheap and fast
  - Most of main-memory data is live

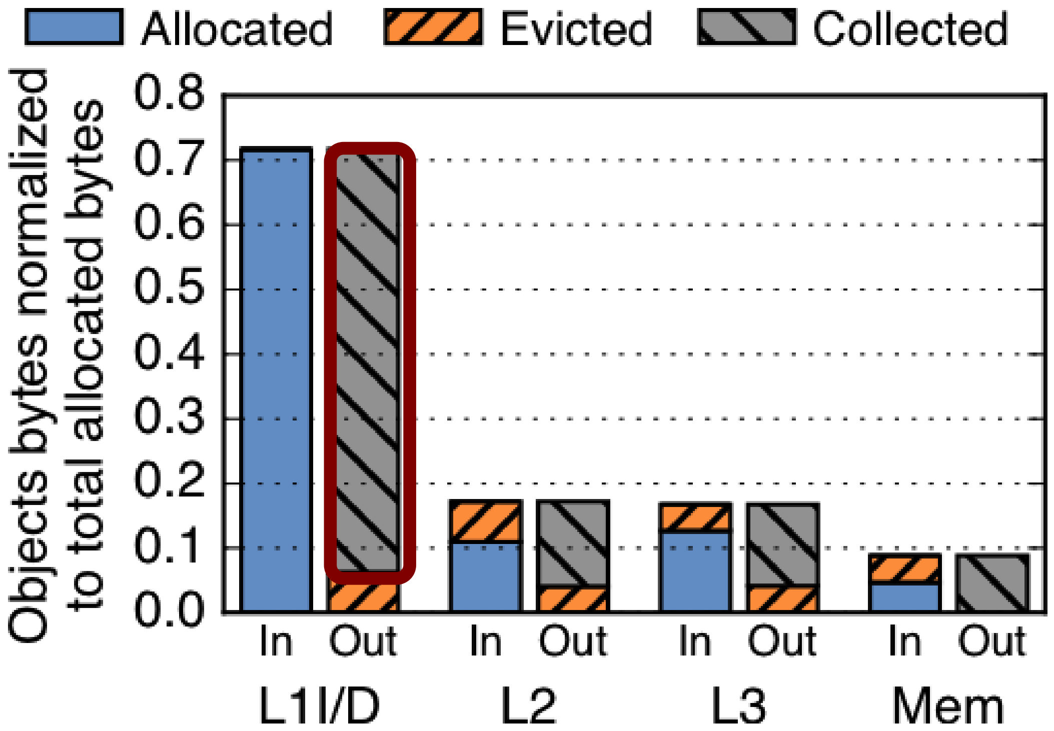
Collection-evictions reduce data movement

- Hotpads unifies the locality principle and the generational hypothesis
- Hotpads acts like a super-generational collector
  - Accesses to short-lived objects are cheap and fast
  - Most of main-memory data is live



Collection-evictions reduce data movement

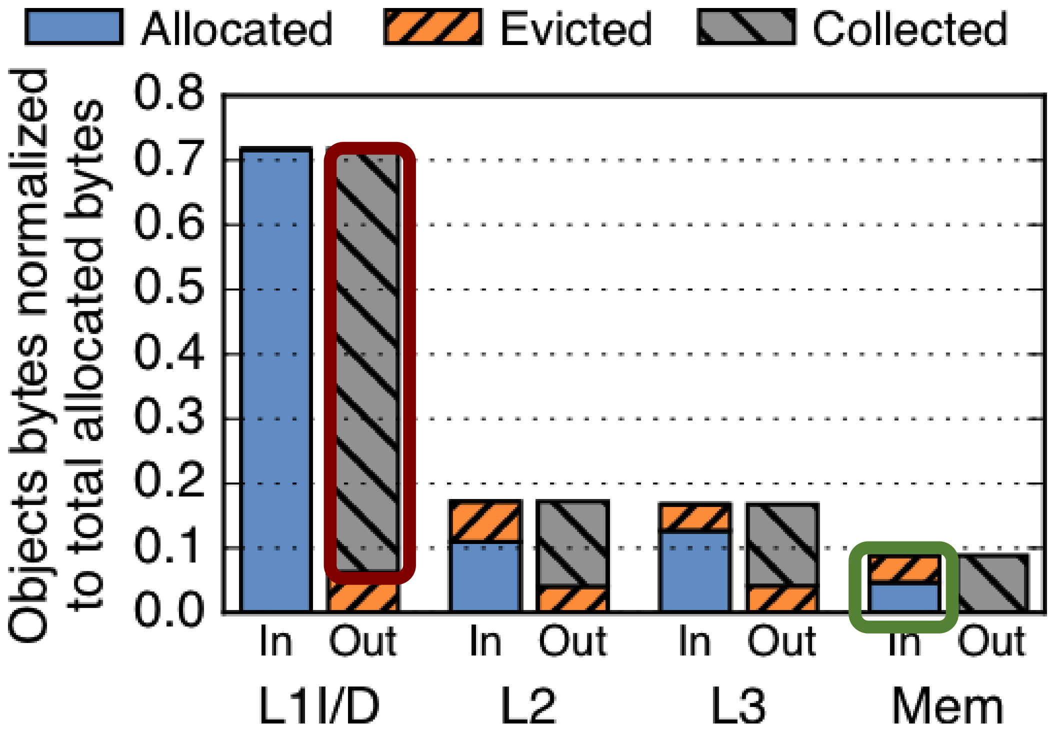
- Hotpads unifies the locality principle and the generational hypothesis
- Hotpads acts like a super-generational collector
  - Accesses to short-lived objects are cheap and fast
  - Most of main-memory data is live



Most objects are collected in the L1 pad

Collection-evictions reduce data movement

- Hotpads unifies the locality principle and the generational hypothesis
- Hotpads acts like a super-generational collector
  - Accesses to short-lived objects are cheap and fast
  - Most of main-memory data is live



Most objects are collected in the L1 pad

90% of object bytes never reach main memory

See paper for additional features

See paper for additional features

- Supporting large objects with subobject fetches

See paper for additional features

- Supporting large objects with subobject fetches
- Object-level pad coherence

See paper for additional features

- Supporting large objects with subobject fetches
- Object-level pad coherence
- Legacy mode to support flat-address-based programs

See paper for additional features

- Supporting large objects with subobject fetches
- Object-level pad coherence
- Legacy mode to support flat-address-based programs
- … and more details!

Evaluation

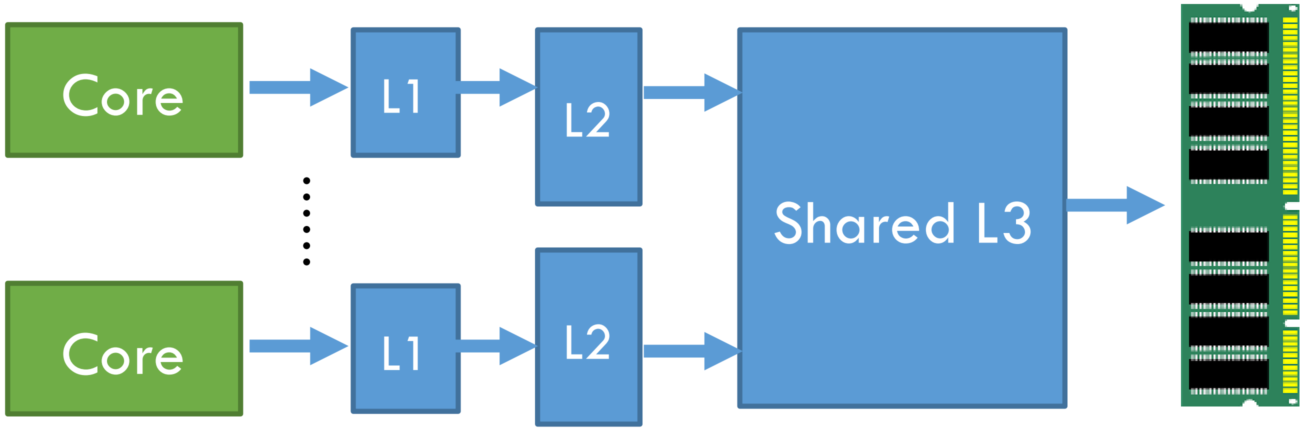
Evaluation

- We simulate Hotpads using MaxSim [Rodchenko et al., ISPASS'17]
  - A simulator combining ZSim and Maxine JVM

Evaluation

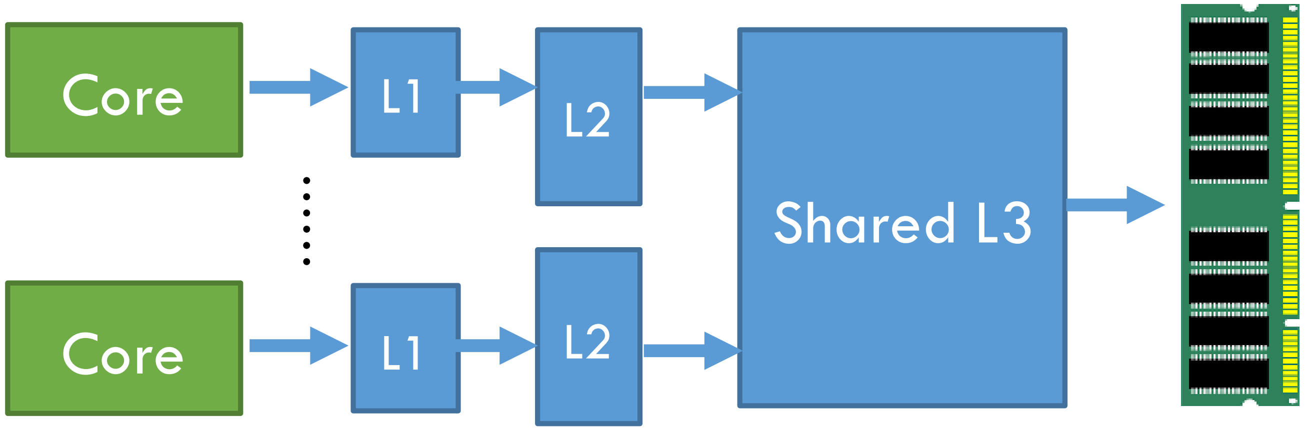
- We simulate Hotpads using MaxSim [Rodchenko et al., ISPASS'17]
  - A simulator combining ZSim and Maxine JVM

- Modeled system
  - 4 000 cores
  - 3-level cache or pad hierarchy



Evaluation

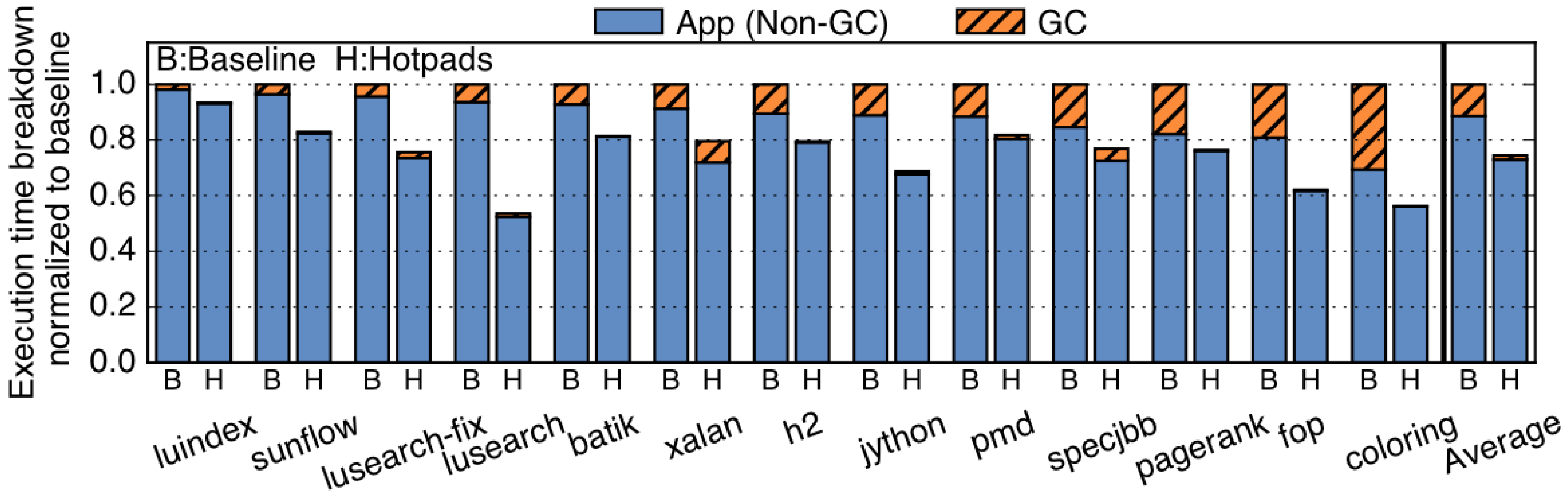
- We simulate Hotpads using MaxSim [Rodchenko et al., ISPASS'17]
  - A simulator combining ZSim and Maxine JVM
- Modeled system
  - 4 000 cores
  - 3-level cache or pad hierarchy



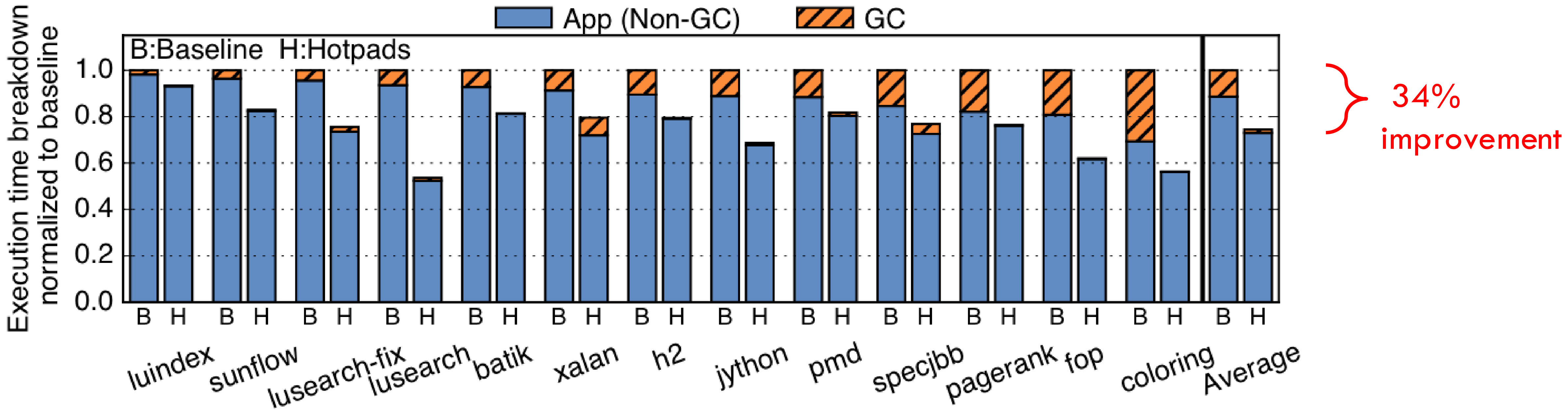
- Workloads
  - 13 Java workloads from Dacapo, SpecJBB, and JgraphT
  - JVM modified to use the Hotpads ISA

Hotpads outperforms conventional hierarchies

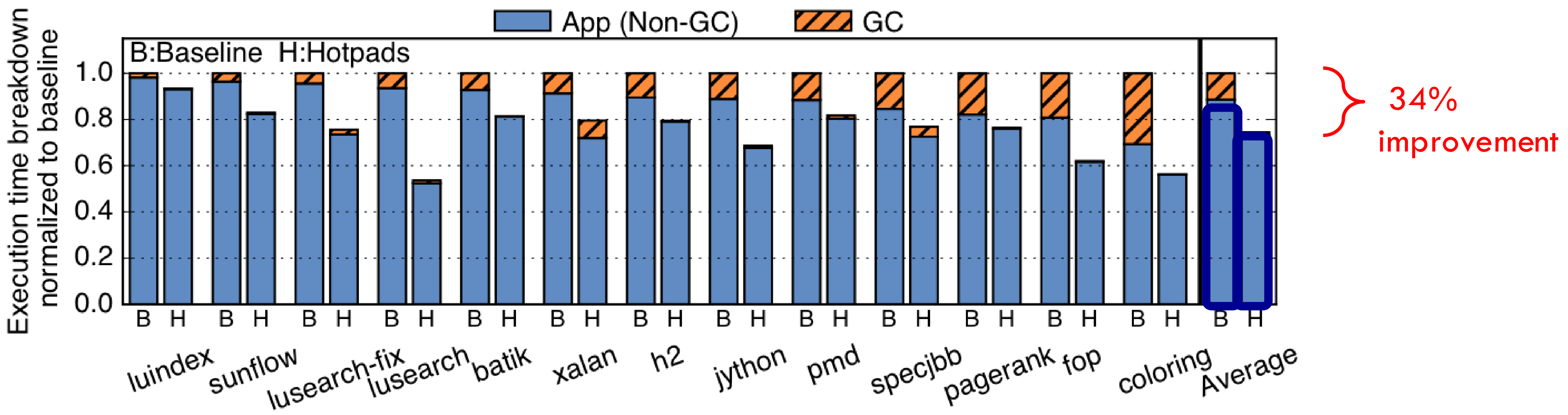
Hotpads outperforms conventional hierarchies



Hotpads outperforms conventional hierarchies

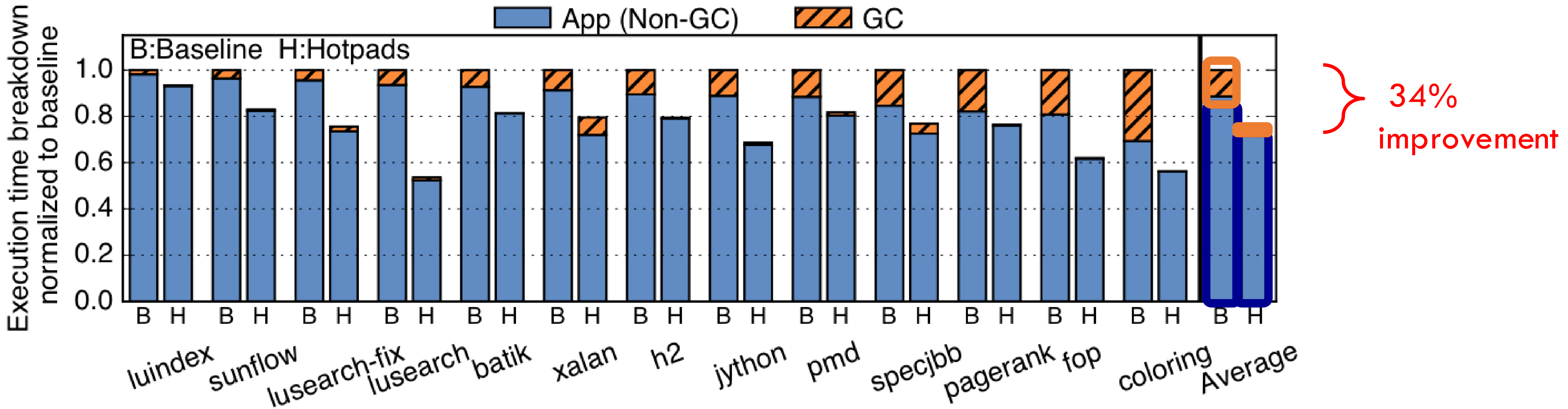


Hotpads outperforms conventional hierarchies



1. In-hierarchy allocation reduces memory stalls in application code

Hotpads outperforms conventional hierarchies

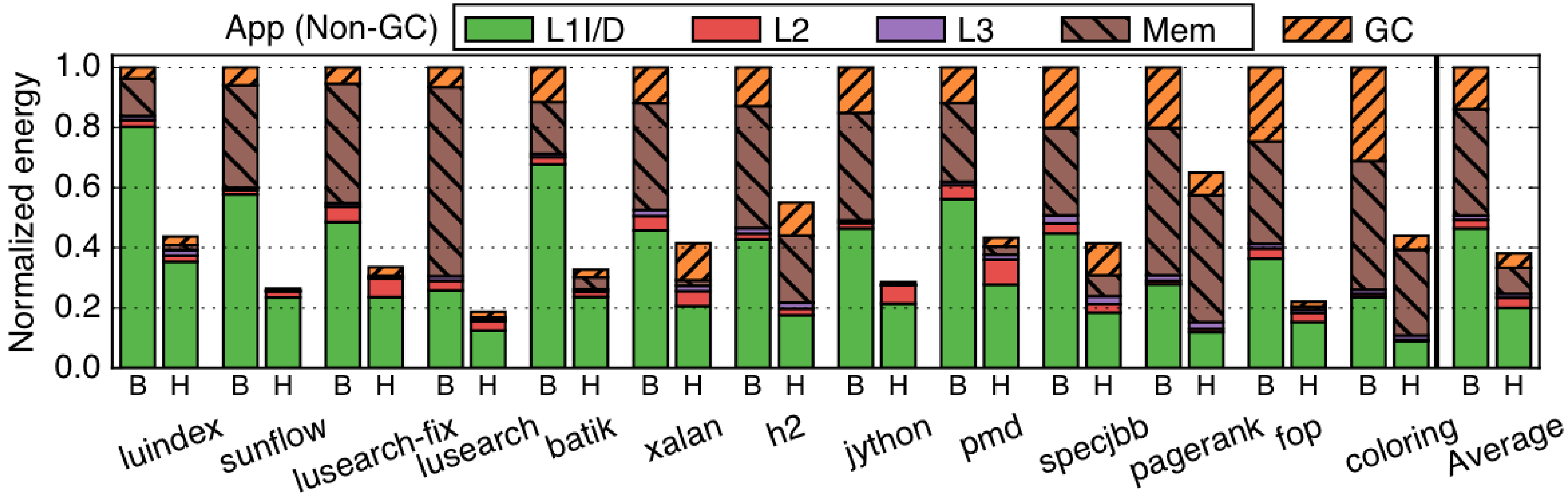


1. In-hierarchy allocation reduces memory stalls in application code

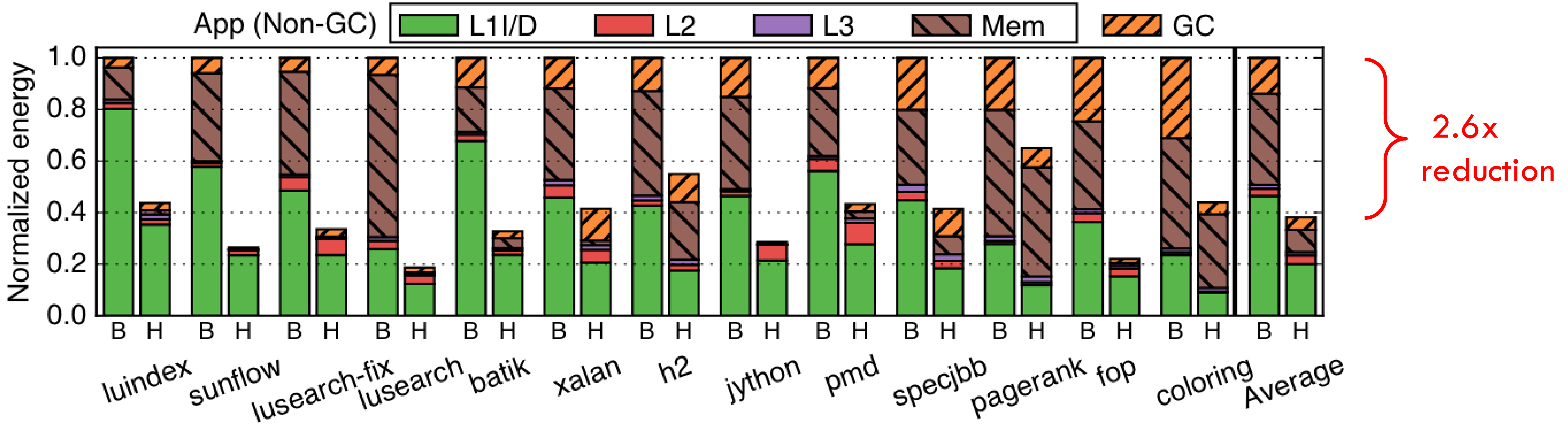
2. Hardware-based collection-evictions reduce GC overheads

Hotpads reduces dynamic memory hierarchy energy

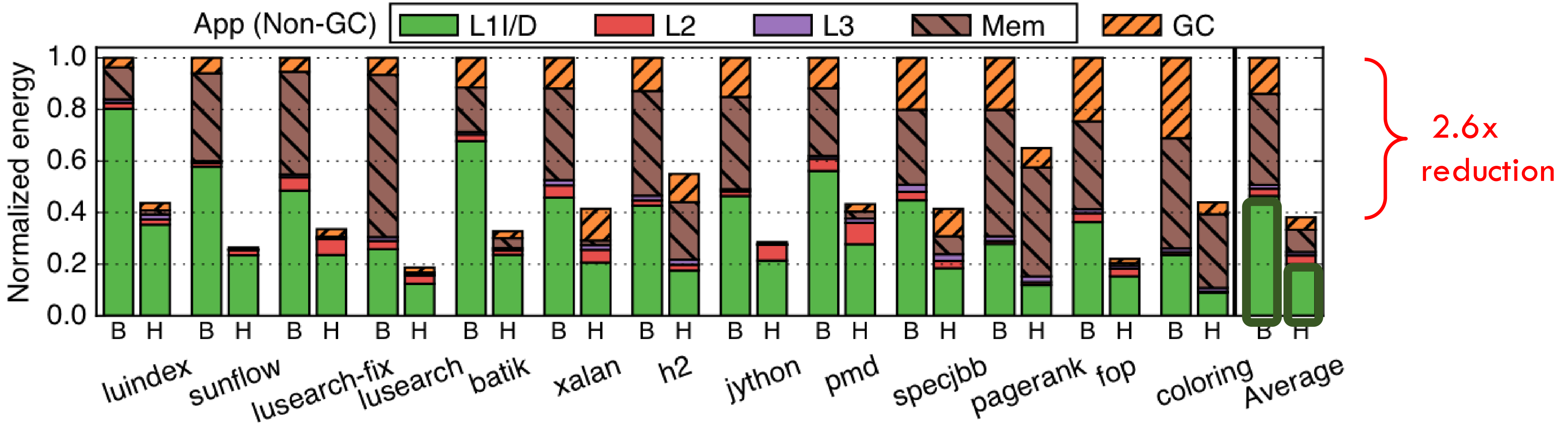
Hotpads reduces dynamic memory hierarchy energy



Hotpads reduces dynamic memory hierarchy energy

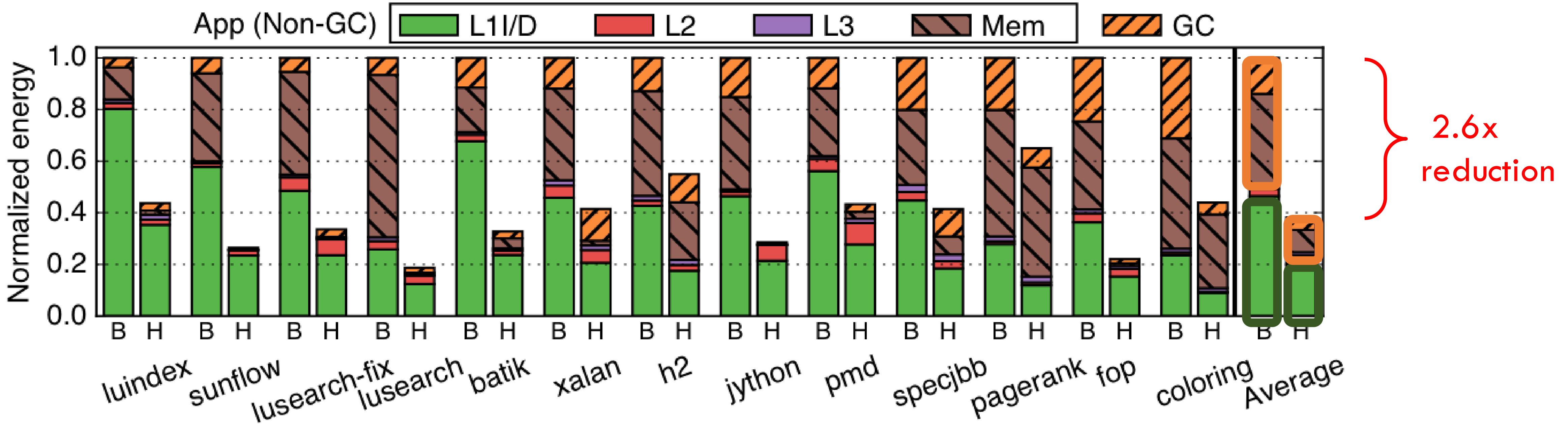


Hotpads reduces dynamic memory hierarchy energy



1. Pointer rewriting and direct accesses reduce L1 energy by 2.3x

Hotpads reduces dynamic memory hierarchy energy



1. Pointer rewriting and direct accesses reduce L1 energy by 2.3x

2. Hierarchical collection-evictions reduce memory and GC energy

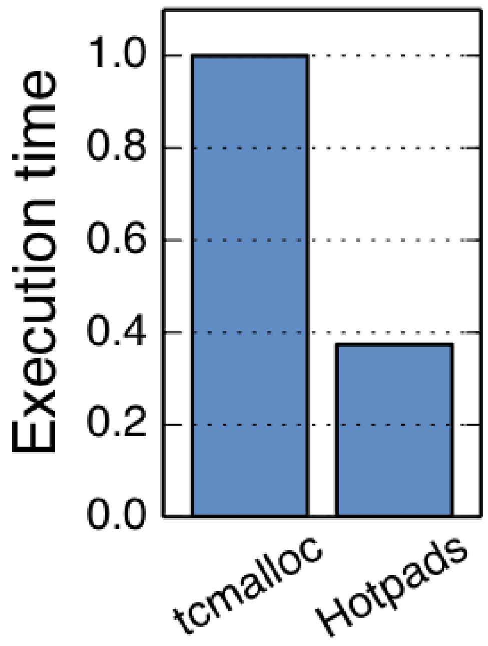
Hotpads also provides benefits on compiled code

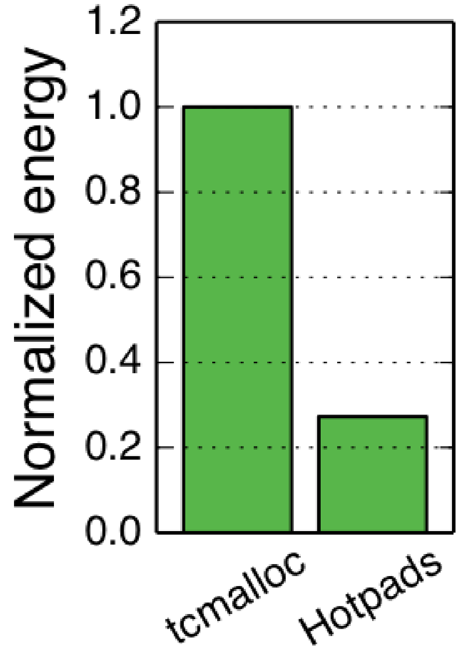
Hotpads also provides benefits on compiled code

- We study an allocation-heavy, binary-tree benchmark written in C
  - Compare Hotpads with tcmalloc, a state-of-the-art memory allocator

Hotpads also provides benefits on compiled code

- We study an allocation-heavy, binary-tree benchmark written in C
  - Compare Hotpads with tcmalloc, a state-of-the-art memory allocator

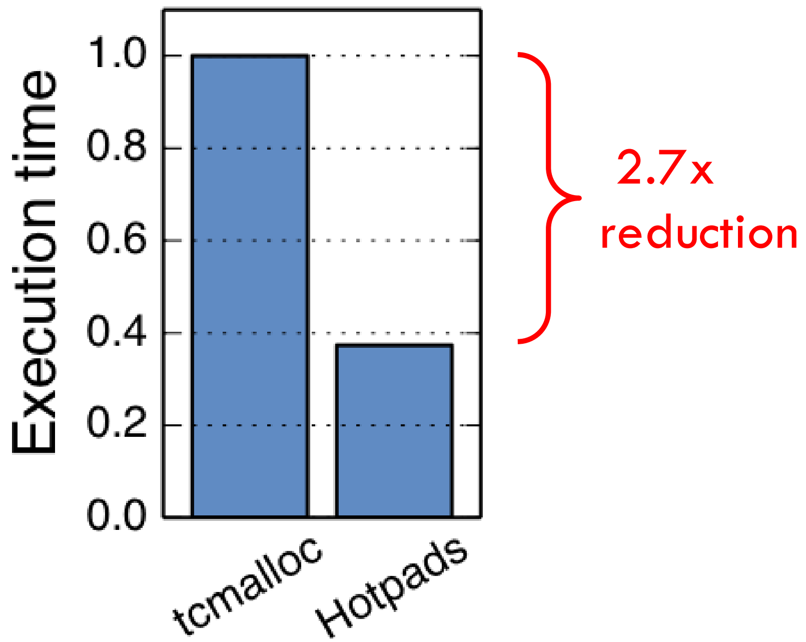


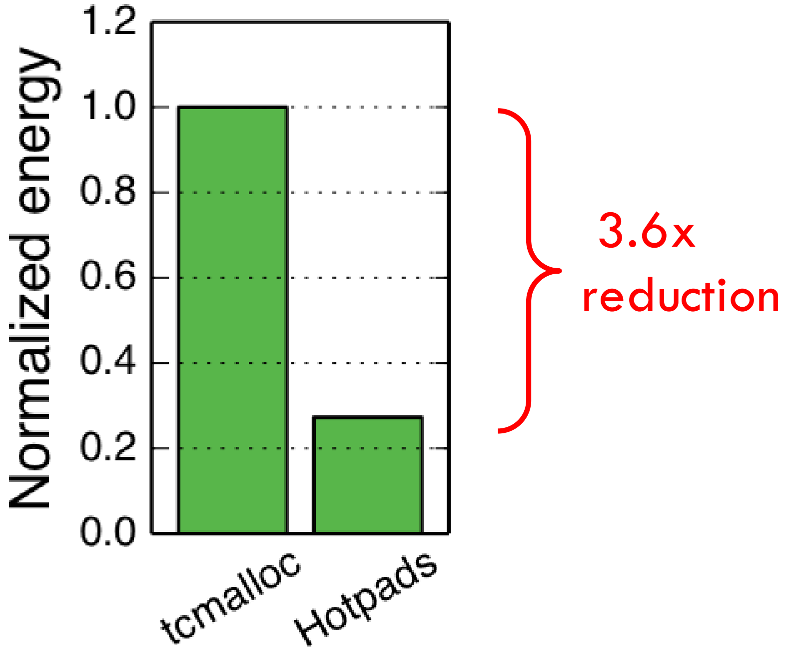


Hotpads improves performance and energy efficiency over manual memory management

Hotpads also provides benefits on compiled code

- We study an allocation-heavy, binary-tree benchmark written in C
  - Compare Hotpads with tcmalloc, a state-of-the-art memory allocator





Hotpads improves performance and energy efficiency over manual memory management

See paper for more results

- Results for multithreaded workloads

- Detailed analysis of pointer rewriting and CEs

- Comparison with other cache-based techniques
  - Enhanced baseline using DRRIP and stream prefetchers
  - Cache scrubbing and zeroing [Sartor et al., PACT'14]

- Legacy mode performance on SPECCPU apps

An object-based memory hierarchy provides tremendous benefits

An object-based memory hierarchy provides tremendous benefits

- Modern programs operate on objects, not cache lines

An object-based memory hierarchy provides tremendous benefits

- Modern programs operate on objects, not cache lines

- Hotpads is an object-based memory hierarchy that supports objects in the ISA and hides the memory layout

An object-based memory hierarchy provides tremendous benefits

- Modern programs operate on objects, not cache lines

- Hotpads is an object-based memory hierarchy that supports objects in the ISA and hides the memory layout

- Hotpads outperforms conventional cache hierarchies because it:
  - Moves objects rather than cache lines
  - Avoids most associative lookups with *pointer rewriting*
  - Provides hardware support for *in-hierarchy allocation* and unified *collection-eviction*

An object-based memory hierarchy provides tremendous benefits

- Modern programs operate on objects, not cache lines
- Hotpads is an object-based memory hierarchy that supports objects in the ISA and hides the memory layout
- Hotpads outperforms conventional cache hierarchies because it:
  - Moves objects rather than cache lines
  - Avoids most associative lookups with *pointer rewriting*
  - Provides hardware support for *in-hierarchy allocation* and unified *collection-eviction*
- Hotpads also unlocks new memory hierarchy optimizations

Thanks! Questions?

- Modern programs operate on objects, not cache lines

- Hotpads is an object-based memory hierarchy that supports objects in the ISA and hides the memory layout

- Hotpads outperforms conventional cache hierarchies because it:
  - Moves objects rather than cache lines
  - Avoids most associative lookups with *pointer rewriting*
  - Provides hardware support for *in-hierarchy allocation* and unified *collection-eviction*

- Hotpads also unlocks new memory hierarchy optimizations